intel®

iSBC™
Applications
Manual

142687

PREFACE

Since Intel introduced the iSBC 80/10 Single Board Computer in early 1976, the family of Intel OEM Microcomputer Systems has grown rapidly. Original equipment manufacturers and volume end-users alike have responded to the concept originated by Intel of having all the functions of a computer — central processing unit, memory, input–output and system expansion capability — present on one printed circuit board.

The capabilities of a single board computer have been enhanced by the creation of the industry-standard MULTIBUS system bus. System expansion boards have been introduced for memory, serial I/O and parallel I/O expansion, as well as analog I/O, DMA controllers and peripheral controllers. A unique feature of the MULTIBUS architecture, however, is its capability to support multiple single board computers. This capability permits sophisticated multiprocessing configurations using standard off-the-shelf 8-bit and 16-bit single board computers. Powerful software tools like the RMX/80 Real-Time Multitasking Executive, the FORTRAN run-time package and the resident BASIC interpreter also are key members of the iSBC product family. They provide users with the tools for quick implementations of simple or complex systems. The recently introduced iCS product line provides chassis and signal conditioning/termination strips as well as board level products which were developed specifically for industrial users.

This application manual is divided into three sections: iSBC Hardware, iSBC Software and iCS Products. It contains all of the current application notes, reliability reports, magazine articles and professional journal reprints on the products of the Intel iSBC product family. We have compiled all of this information into a single publication for your convenience. Please contact us with your questions, comments, and suggestions on how we may provide you with useful information on our products.

INTEL CORPORATION
OEM Microcomputer Systems
Applications Engineering
Hillsboro, Oregon 97123

FUNCTIONAL INDEX

1 iSBC Hardware

iSBC HARDWARE

INTRODUCTION

The current hardware products available in the iSBC product line include six single board computers and over 30 expansion boards and accessories. Each Intel single board computer provides all the resources of a full computer (i.e., CPU, read/write memory, read only, parallel I/O and serial I/O) on a single printed circuit board. The iSBC 655 and iSBC 660 chassis extend these capabilities into low cost, fully packaged RETMA rack-mountable computers. The Intel single board computers are supported by a complete line of memory, parallel and serial I/O, digital I/O and analog I/O expansion boards, and peripheral and DMA controllers, all of which are compatible with the industry-standard microcomputer bus — the Intel MULTIBUS system bus.

This section contains application notes and magazine articles covering the architectural features of the iSBC product family, the iSBC 80/10A, iSBC 80/30, and iSBC 86/12A Single Board Computers, and the iSBC 544 Intelligent Slave Board. In addition, reliability reports for the iSBC 80/10A and iSBC 86/12A boards are included.

TABLE OF CONTENTS

iSBC 80/10A — SYSTEM 80/10 Single Board Computer Applications

Thomas Rolander
Microcomputer Applications

iSBC 80/10A–SYSTEM 80/10 Single Board Computer Applications

Contents

INTRODUCTION

The recent entry of the single board computer into the broad field of electronic applications is substantiating the billing as a "super component". Single board computers provide a solution to several problems that have not been solved by the use of conventional computers: cost, size, and design specialization.

Many potential microcomputer applications have been overlooked because of the design tasks required to build a microcomputer system. These tasks traditionally include interfacing of the system clock, read/write memory, I/O ports and drivers, serial communications interface, bus control logic and drivers. Intel's iSBC 80/10A enables the design engineer to concentrate on the application of microcomputers, rather than on implementation details.

This application note begins with an overview of the Intel® iSBC 80/10A. Readers who are familiar with the iSBC 80/10A may choose to skip to the applications section, which describes the following typical iSBC 80/10A applications:

- The iSBC 80/10A used for instrumentation control of a Fluke 8375 Digital Multimeter.
- The iSBC 80/10A used as a SCADA Terminal in a communication application.
- The iSBC 80/10A used for temperature monitoring in a process control application.
- The iSBC 80/10A used as an interrupt driven device controller for a Centronics printer.

Each example shows the user program and hardware required for the application. The program listings are interspersed with the text describing the application. Both 8080 Macro Assembly Language and Intel's PL/M-80 are used in the examples.

The software was developed on an Intel® Microcomputer Development System (MDS). The MDS provided the tools necessary to edit, assemble or compile, link and locate the application software. Hardware development was facilitated by the use of Intel's In-Circuit Emulator (ICE 80). For further information regarding the Microcomputer Development System, the reader is referred to the publications listed at the beginning of this application note.

OVERVIEW

The iSBC 80/10A is a member of Intel's complete line of OEM computer systems which take full advantage of Intel's LSI technology to provide economical, self-contained computer based solutions for OEM applications. The iSBC 80/10A is a complete computer system on a single 6.75-by-12 inch printed circuit card. A block diagram of the iSBC 80/10A is shown in Figure 1.

Intel's powerful 8-bit n-channel MOS 8080A CPU, fabricated on a single LSI chip, is the central processor for the iSBC 80/10A. The 8080A contains six 8-bit general purpose registers and an accumulator. The six general purpose registers may be addressed individually or in pairs, providing both single and double precision operators.
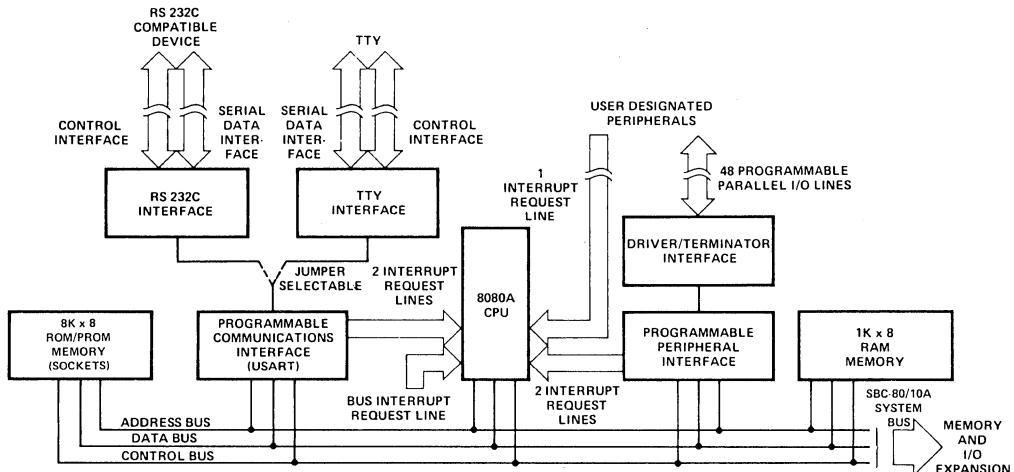
1. Interrupts originating from the Programmable Communications Interface and Programmable Peripheral Interface are jumper selectable.

Figure 1. iSBC 80/10A Block Diagram

The 8080A has a 16-bit program counter which allows direct addressing of up to 64K bytes of memory. An external stack, located within any portion of read/write memory, may be used as a last in/first out stack to store the contents of the program counter, flags, accumulator and all of the six general purpose registers. A 16-bit stack pointer addresses the external stack. This provides subroutine nesting that is bounded only by memory size.

The iSBC 80/10A contains 1K bytes of read/write memory using Intel's low power static RAM. All on board RAM read and write operations are performed at maximum processor speed. Four sockets for up to 8K bytes of non-volatile read-only memory are provided on the board. Read-only memory may be added in 1K byte increments (up to 4K total) using Intel® 8708 erasable and electrically reprogrammable ROMs (EPROMs) or Intel 8308 masked ROMs. Optionally, if more than 4K bytes are required, read only memory may be added in 2K byte increments (up to 8K total) using Intel® 2716 EPROMs or 2316E masked ROMs. All on-board ROM or EPROM read operations are performed at maximum processor speed.

The iSBC 80/10A contains 48 programmable parallel I/O lines implemented using two Intel® 8255 Programmable Peripheral Interfaces. The system software is used to configure the I/O lines in any combination of unidirectional input/output, and bidirectional ports indicated in Table I. Therefore, the I/O interface may be customized to meet specific peripheral requirements. To support the large number of possible I/O configurations, sockets are provided for interchangeable I/O line drivers and terminators. Hence, the I/O interface provides the appropriate combination of optional line drivers and terminators to allow the required sink current, polarity, and drive/termination characteristics for each application. The 48 programmable I/O lines and signal ground lines are brought out to two 50-pin edge connectors that mate with flat, round, or woven cable.

A programmable communications interface using Intel's 8251 Universal Synchronous/Asynchronous Receiver/Transmitter (USART) is contained on the iSBC 80/10A. A jumper selectable baud rate generator provides the 8251 with all common communication frequencies. The 8251 can be programmed by the user's system software to select the desired asynchronous or synchronous serial data transmission technique (including IBM Bisync). The mode of operation (synchronous or asynchronous), data format, control character format, parity, and asynchronous transmission rate are all under program control. The 8251 provides full duplex, double buffered transmission and receive capability. Parity, overrun, and framing error detection circuits are all incorporated in the 8251. The inclusion of jumper selectable TTY or EIA RS232C compatible interfaces on the board, in conjunction with the 8251, provide a direct interface to teletypes, CRTs, asynchronous and synchronous modems, and other RS232C compatible devices. The RS232C or TTY command lines, serial data lines, and signal ground lines are brought out to a 25-pin edge connector that mates with RS232C compatible flat, round, or woven cable.

Interrupt requests may originate from six sources. Two from the 8255's, two from the 8251 and two from user designated peripheral devices.

TABLE 1 INPUT/OUTPUT PORT MODES OF OPERATION

PORT	NO. OF LINES	MODE OF OPERATION					
		UNIDIRECTIONAL				BIDIRECTIONAL	CONTROL
		INPUT		OUTPUT			
		UNLATCHED	LATCHED & STROBED	LATCHED	LATCHED & STROBED		
1	8	X	X	X	X	X	
2	8	X	X	X	X		
3	8	X		X			X[1]
4	8	X		X			
5	8	X		X			
6	4	X		X			
	4	X		X			

1. Note: Port 3 must be used as a control port when either Port 1 or Port 2 are used as a latched and strobed input or a latched and strobed output or Port 1 is used as a bidirectional port.

The 8255's can generate interrupts when a byte of information is ready to be transferred to the CPU (i.e., input buffer full) or a byte of information has been transferred to a peripheral device (i.e., output buffer is empty).

The 8251 can generate interrupts when a character is ready to be transferred to the CPU (i.e., receive channel buffer is full) or a character is ready to be transmitted (i.e., transmit channel data buffer is empty).

The user designated peripheral devices can generate two interrupts: one via the system bus and the other via the I/O edge connector.

The two interrupts from the 8255's and the two interrupts from the 8251 are all individually maskable under program control. The six interrupt request lines share a single CPU interrupt level. When an interrupt request is recognized, a RESTART 7 instruction is generated. The processor responds by suspending program execution and making a subroutine call to a user defined interrupt service routine originating at location 38 (Hexadecimal).

iSBC 80/10A memory and I/O capacity may be increased by adding standard Intel memory and I/O boards. Modular expandable backplanes and card cages, each with a four-board capacity, are available to support multi-board systems.

The development cycle of iSBC 80/10A based products may be significantly reduced using the Intellec Microcomputer Development System. The resident macro-assembler, PL/M-80 compiler, text editor, and system monitor greatly simplify the design, development, and debug of user designed iSBC 80/10A system software. A diskette-based system allows programs to be loaded, assembled, edited, and executed faster than using conventional paper tape, card, or cassette peripherals. A unique In-Circuit Emulator (ICE 80) provides the capability of developing and debugging software directly on the iSBC 80/10A.

iSBC CONFIGURATION OPTIONS

The iSBC 80/10 provides the user with a powerful and flexible I/O capability for both parallel and serial transfers. This section discusses the user programmable and jumper-selectable options, and bus interfacing.

SERIAL I/O OPTIONS

The serial I/O interface, using Intel's 8251 USART, provides a serial data communications channel that can be programmed to operate with most of the current serial data transmission protocols. There are three general areas of serial I/O options:

1. choice of interface type, RS232C or current loop,

2. baud rate and program-selectable mode options,

3. choice of an interrupt mechanism.

The user has the choice, through jumper connections, of configuring the serial I/O logic to present either an RS232C or a 20 mA current loop interface to an external device. If an RS232C interface is used, the 8251 can assume the role of a "data set" or a "data processing terminal". This enables the serial interface to be connected to different devices such as modems and computer terminals.

There are two factors which enter into the choice of baud rate. They are the actual clock frequency used to drive the transmit/receive clocks on the 8251 and the baud rate factor selected by a programmable mode instruction control word output by the processor to the 8251. The baud rate factor is used to effectively divide the 8251 transmit and receive clocks by 1, 16 or 64. During normal operation a factor of 16 is selected for asynchronous transmissions from 9.6K to 300 baud. A factor of 64 must be used to achieve a baud rate of 110. The baud rate factor is only applicable to asynchronous transmission, as all synchronous transmission is done with an implied factor of one.

Before beginning serial I/O operations, the 8251 must be program-initialized to support the desired mode of operation. The CPU initializes the 8251 by issuing a set of control bytes to the USART device. These control words specify:

- synchronous or asynchronous operation
- baud rate factor
- character length
- number of stop bits
- even/odd parity
- parity/no parity

Refer to the *iSBC 80/10 and iSBC 80/10A Single Board Computer Hardware Reference Manual* or the "8251 Application Note" for details on the control words used to direct the operation of the 8251.

The serial I/O logic can be configured with different forms of interrupt request mechanisms. By connecting a jumper, the user can allow the 8251's Receiver Ready output to generate an interrupt request. The Receiver Ready output goes high whenever the Receiver Enable bit of the command

word has been set and the 8251 contains a character that is ready to be input to the CPU. The user can also choose to have the 8251's Transmitter Ready or Transmitter Empty output activate the interrupt request. The Transmitter Empty goes high when the 8251 has no characters to transmit. Transmitter Ready is high when the 8251 is ready to accept a character from the CPU. Both Transmitter Empty and Transmitter Ready are enabled by setting the Transmit Enable bit of the command word. Upon receiving an interrupt, the program can determine the actual condition which is responsible for the interrupt by reading the status of the 8251 device.

PARALLEL I/O OPTIONS

The parallel I/O interface consists of six 8-bit I/O ports implemented with two Intel 8255 Programmable Peripheral Interface devices. Eight lines already have a bidirectional driver and termination network permanently installed. The remaining 40 lines are uncommitted. Sockets are provided for the installation of active driver networks or passive termination networks as required to meet the specific needs of the user system.

The primary considerations in determining how to use each of the six I/O ports are:

1. choice of operating mode,
2. direction of data flow (input, output or bidirectional),
3. selection of interrupt mechanism,
4. choice of driver/termination networks for the port's data path.

Operating Modes. There are three basic operating modes that can be selected by the system software. The modes of operation will be described here in general terms, leaving it to the reader to obtain details from the *iSBC 80/10 and iSBC 80/10A Single Board Computer Hardware Reference Manual* or the "8255 Application Note."

Mode 0 is a basic input/output functional configuration which provides simple input and output operations. No "handshaking" is required, data is simply written to or read from a specified port. The outputs are latched and the inputs are unlatched.

Mode 1 is a strobed input/output functional configuration which provides a means for transferring I/O data to or from a specified port in conjunction with strobes or handshaking signals. The outputs are latched and are accompanied by

an output control line which indicates that the processor has loaded the output port with a data byte. The input data is latched when accompanied by its externally operated control signal.

Mode 2 is a strobed bidirectional bus input/output functional configuration which provides a means for communicating with a peripheral device or structure on a single 8-bit bus for both transmitting and receiving data. Handshaking signals are provided to maintain proper bus flow discipline in a manner similar to mode 1.

Data Flow Direction. In addition to the choice of operating mode, the user may also specify the direction of data flow, input or output from the 8255's. At the time of RESET, the 8255's are configured into the input mode until altered by a control word directed to the control word register. When an output mode control word is received, all of the data bits are set to the low output state.

Interrupt Mechanism. When the 8255 is programmed to operate in mode 1 or mode 2, control signals are provided that can be used as interrupt request inputs to the CPU. The interrupt request signals, generated from one of the ports (port C), can be inhibited or enabled by setting or resetting the associated interrupt enable flip-flop, using the bit set/reset function of port C. This function allows the programmer to mask the interrupts from specific I/O devices without affecting any other device in the interrupt structure.

Driver/Termination Networks. Depending on the direction of data flow, the user will select the appropriate TTL line drivers and Intel terminators that are compatible with the I/O driver/terminator sockets on the iSBC 80/10A. The list of suitable line drivers includes those with inverting, non-inverting, and open collector characteristics. There are two types of terminators: a 220-ohm/330-ohm divider or a 1K ohm pull-up.

BUS INTERFACING

The system bus interface logic consists of three general groups of circuitry:

1. gates that accept the various bus control signals, the interrupt request lines, and the ready indications, and then apply these signals to the CPU logic elements,
2. the system bus drivers,
3. the failsafe circuitry which generates an acknowledgment during interrupt sequences and during those cycles in which an ac-

knowledgment is not returned because a non-existent device was inadvertently addressed.

Bus Interface Signals. The following paragraphs describe portions of the system bus interfacing logic relevant to interfacing a user device to the iSBC 80/10A. (Note: Whenever a signal is active-low, its mnemonic is followed by a slash; for example, MRDC/ means that the level on that line will be low when the memory read command is true.)

BCLK/ — Bus clock; used to synchronize bus control circuits on all master modules. BCLK/ has a frequency of 9.216 MHz. BCLK/ may be slowed, stopped or single stepped, if desired.

INIT/ — Initialization signal; resets the entire system to a known internal state.

BPRN — Bus priority input signal; indicates to the iSBC 80/10A that a higher priority master module is requesting use of the system bus. BPRN suspends the processing activity and drivers of the iSBC 80/10A until the signal goes low.

BUSY/ — Bus busy signal; indicates that the bus is currently in use. BUSY/ prevents all other master modules from gaining control of the bus. BUSY/ is driven by the HLDA/ output from the iSBC 80/10A in response to a BPRN input. It indicates that the bus is available.

MRDC/ — Memory read command; indicates that the address of a memory location has been placed on the system address lines and specifies that the contents of the addressed location are to be read and placed on the system data bus.

MWTC/ — Memory write command; indicates that the address of a memory location has been placed on the system address lines and that a data word has been placed on the system data bus. MWTC/ specifies that the data word is to be written into the addressed memory location.

IORC/ — I/O read command; indicates that the address of an input port has been placed on the system address bus and that the data at that input port is to be read and placed on the system data bus.

IOWC/ — I/O write command; indicates that the address of an output port has been placed on the system address bus and that the contents of the system data bus are to be output to the addressed port.

XACK/ — Transfer acknowledge signal; the required response of an external memory location or I/O port which indicates that the specified read/write operation has been completed (that is, data has been placed on, or accepted from, the system data bus lines).

AACK/ — An advance acknowledge, in response to a memory read or write command, that allows the memory to complete the specified operation without requiring the CPU to wait.

CCLK/ — Constant clock; provides a clock signal of constant frequency (9.216 MHz) for use by optional memory and I/O expansion boards. The same signal is used to drive both CCLK/ and BCLK/.

INTR1/ — Externally generated interrupt request.

ADR0/–ADRF/ — 16 Address lines; used to transmit the address of the memory location or I/O port to be accessed. ADRF/ is the most significant bit.

DAT0/–DAT7/ — Bidirectional data lines; used to transmit/receive information to/from a memory location or I/O port. DAT7/ is the most significant bit.

Bus Acknowledges. Further distinction between transfer acknowledge (XACK/) and advance acknowledge (AACK/) is required. All external memory and I/O transfer requests must return XACK/ to the iSBC 80/10A (even if AACK/ is also returned). XACK/ indicates that data has been placed on (read command) or accepted from (write command) the system data bus lines. AACK/ is an advance acknowledge in response to a memory or I/O port command. It has been provided because the 8080A samples the ready line before valid data is required on the bus. If this condition is properly anticipated, AACK/ can be returned before the data is actually read, thus allowing an earlier operation to be completed. AACK/ should be used only with a thorough understanding of the additional information provided in the *iSBC 80/10 and iSBC 80/10A Single Board Computer Hardware Reference Manual. DMA Transfers.* An external device can make DMA transfers to or from RAM expansion boards. The transfer is coordinated with the iSBC 80/10A by means of two bus signals: bus priority input (BPRN) and bus busy (BUSY/). The first step in making a DMA transfer is to obtain control of the system bus. This is

achieved by asserting BRPN to the iSBC 80/10A and then waiting until the iSBC 80/10A returns BUSY/, indicating that it has relinquished control of the system bus. When this step is completed the external device may proceed with its DMA transfers until it is finished. At that time BPRN should be removed to allow the iSBC 80/10A to regain control of the system bus. It should be noted that the iSBC 80/10A is placed in a hold state when it does not have control of the system bus.

APPLICATIONS

The iSBC 80/10A may be applied to a wide variety of applications. Specific applications in four areas are presented in this application note. They are presented to illustrate a broad spectrum of single board computer capabilities and to demonstrate the use of various system features.

INSTRUMENTATION

Microprocessors have been used in instrumentation for many tasks ranging from handling simple interface functions to control of the analog to digital conversion process. The use of a single board computer can further serve in the application of instruments themselves to laboratory or process control environments. It is quite often necessary in these applications to control instrumentation remotely. A number of rather expensive minicomputer-controlled solutions now exist on the market as automatic test equipment (ATE) systems. The iSBC 80/10A presents itself as a cost effective solution in situations where the larger ATE systems are beyond economic justification.

The iSBC 80/10A can be the sole CPU element in the system, providing instrumentation control and computational capability; or it can supplement a larger host CPU by handling distributed processing requirements.

Instrumentation Control Application Example

Most instruments such as DVMs, counters, data loggers, synthesizers, etc., have optional data output units (DOUs) and/or remote control units (RCUs). It is particularly time consuming to interface each instrument's DOU/RCU with custom-digital logic. Until the recent IEEE-488 interface standard, there was little in common from one interface to the next. The parallel I/O lines of the iSBC 80/10A provide a common interface element that can be adapted to a majority of the DOUs and RCUs available today by means of software.
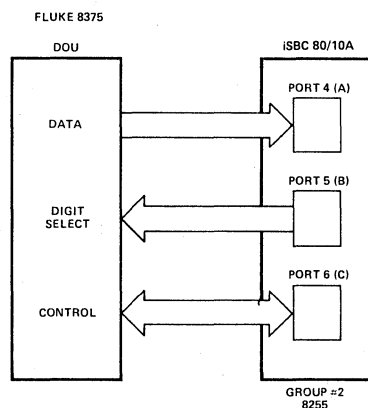
Figure 2. Interface Block Diagram

This instrumentation control application shows how the iSBC 80/10A has been used to control and read the data from the data output unit (DOU) of a Fluke 8375 Digital Multimeter.

Interfacing the iSBC 80/10A to the Fluke 8375 DOU has been accomplished through the use of three parallel I/O ports shown in Figure 2. An 8-bit port has been used to read input data from the Fluke 8375 DOU. Another 8-bit port has been used to control the multiplexing of data from the DOU to the iSBC 80/10A. And, an 8-bit port has been used to provide the required control and monitoring of the following DOU functions: busy flag, sample sync flag, timeout enable, external trigger and trigger inhibit.

The following listing contains a complete program to provide the necessary interface control functions as well as an exercise program. The program listing is interspersed with text that is used to clarify the elements of the program.

```
0 ;
1 ;     INSTRUMENTATION CONTROL APPLICATION
2 ;
3 ;        FLUKE 8375 DIGITAL MULTIMETER
4 ;
5 ;     DATA OUTPUT UNIT (DOU) CONTROLLER
6 ;
7 ;
8
```

The CSEG directs the ISIS-II 8080 Assembler to generate a relocatable code segment. Relocatable code can later be placed at any memory address by Intel's LOCATE program. This lets you write your program without worrying about the application's final memory configuration.

```
 9 ;
10 ;
11 ;        CSEG
12 ;
13
```

Equate Table. The following table is used to give symbolic names to the binary I/O port addresses. The names used later in the program increase readability.

```
14  ;
15  ;
16  ;   EQUATE TABLE
17  ;
18 CWR    EQU    OEBH    ; 8255 #2 CONTROL WORD REGISTER
19 DATIN  EQU    OE8H    ; DECADE PAIR DATA INPUT PORT
20 STB    EQU    OE9H    ; STROBE OUTPUT PORT
21 FLG    EQU    OEAH    ; FLAG INPUT PORT
22 TRG    EQU    OEAH    ; TRIGGER OUTPUT PORT
23  ;
24
```

The exercise program uses some of the subroutines provided in the iSBC 80/10A System Monitor PROMs. The addresses of the subroutines are included in the equate table.

```
25  ;
26  ;
27 GETCH  EQU    0220H   ; GET CONSOLE INPUT, MASK OFF PARITY
28 CO     EQU    01E8H   ; CONSOLE OUTPUT
29 CROUT  EQU    01F3H   ; PRINT <CR><LF>
30 NMOUT  EQU    02C2H   ; DISPLAY BYTE IN ACCUM
31  ;
32
```

The use of the iSBC 80/10A parallel I/O ports requires that the mode of operation be defined for each port. This is typically done by an initialization subroutine executed when the iSBC 80/10A is powered up or reset.

8255 Control Word. When the opcode field (bit 7) of a control word directed to an 8255 is equal to one, the control word is interpreted as a mode definition control word. The mode definition control word format is shown below:
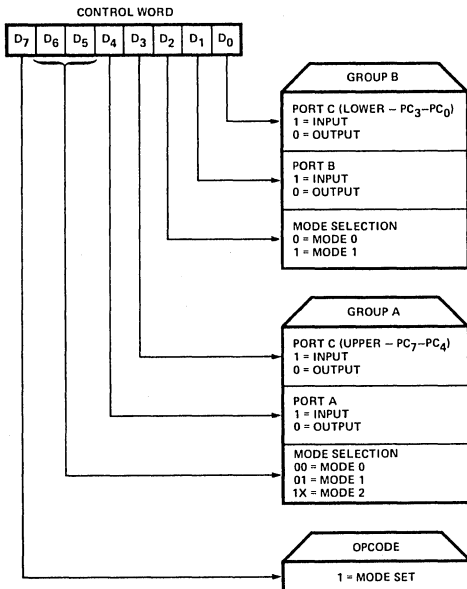
Observing the schematic for the iSBC 80/10A — Fluke 8375 DOU (Figure 3), it can be seen that the 8255 #2 should be configured through the use of the mode control word as:

Port 4 (A)   Mode 0 Input
Port 5 (B)   Mode 0 Output
Port 6 (C)   Bits PC2–PC0 Output
Port 6 (C)   Bits PC5–PC4 Input

The following mode control word is used:

Mode Control Word = 1001 1000 Binary = 98H
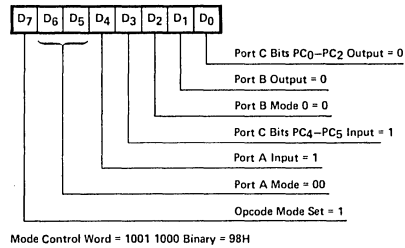
```
33  ;
34  ;
35  ; *** 8255 #2 INITIALIZATION SUBROUTINE
36  ;
37 INIT:
38         MVI    A,10011000B   ; LD MODE CONTROL WORD
39         OUT    CWR           ; OUTPUT TO 8255#2 CNTL WD REG
40  ;
41
```

This coding loads the mode control word into the 8255 #2 control word register. Additional initialization code is required to set the strobe and trigger output ports to an inactive state. The schematic shows that inverting drivers have been used for both the strobes and the trigger. When a command is issued to place port 5 (B) into the output mode, bits PB7–PB0 are set to the low output state. Because the low outputs are then inverted and used as strobes to the Fluke 8375, they must then be disabled. The initialization subroutine concludes by disabling the strobes and trigger. The strobes are signals to the DOU which enable its drivers to send data to the iSBC 80/10A. The trigger is a signal to the DOU that the Fluke 8375 should take a reading.

```
42  ;
43  ;
44         MVI    A,OFFH        ; LD MASK TO:
45         OUT    STB           ;     DISABLE STROBES
46         OUT    TRG           ;     DISABLE TRIGGER
47         RET
48  ;
49
```

External Trigger Control. Two subroutines are implemented to enable and disable the external trigger mode of the instrument. These subroutines use the bit set/reset capability of the 8255 to independently set or reset three control lines of the Fluke 8375 DOU.

When the opcode field (bit 7) of an 8255 control word equals zero, the control word is a port 6 (C) bit set/reset command word.

The bit set/reset control word format is shown below:
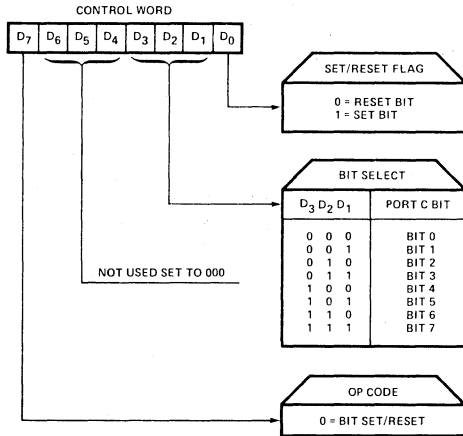
CONTROL WORD

The following example demonstrates how the port 6 (C) bit set/reset control word is constructed to disable the Fluke 8375 external trigger. Note from the schematic (Figure 3) that port 6 (C) bit 0 controls the inhibit external trigger line.

The control word for set Port C bit 0 is 0000 0001 Binary = 01H
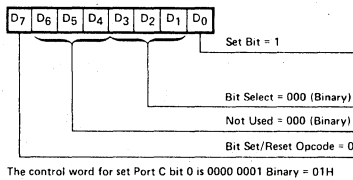
```
50
51 ;
52 ; *** ENABLE EXTERNAL TRIGGER SUBROUTINE ***
53 ;
54 ETRIG:
55        MVI     A,00000000B     ; LD RESET BIT 0 CONTROL WORD
56        OUT     CWR             ; OUTPUT TO 8255#2 CNTL WD REG
57        RET
58 ;
59 ; *** DISABLE EXTERNAL TRIGGER SUBROUTINE ***
60 ;
61 DTRIG:
62        MVI     A,00000001B     ; LD SET BIT 0 CONTROL WORD
63        OUT     CWR             ; OUTPUT TO 8255#2 CNTL WD REG
64        RET
65 ;
66
```

Subroutines to enable and disable the timeouts are written in an analogous fashion. The timeout enable line is controlled by port 6 (C) bit 2.

```
67
68 ;
69 ; *** ENABLE TIMEOUTS SUBROUTINE ***
70 ;
71 EPOS:
72        MVI     A,00000101B     ; LD SET BIT 2 CONTROL WORD
73        OUT     CWR             ; OUTPUT TO 8255#2 CNTL WD REG
74        RET
75 ;
76 ; *** DISABLE TIMEOUTS SUBROUTINE ***
77 ;
78 DPOS:
```

```
79        MVI     A,00000100B     ; LD RESET BIT 2 CONTROL WORD
80        OUT     CWR             ; OUTPUT TO 8255#2 CNTL WD REG
81        RET
82 ;
83
```

Obtaining Readings. The Fluke 8375 DOU allows readings to be taken in one of two modes. The first, a triggered mode, assumes that the external triggering has not been inhibited and requires the positive edge of a pulse with a minimum width of 1 microsecond on the trigger input. Setting and resetting the port 6 (C) bit 1 produces the 8375 external trigger. After a reading is triggered the 8375 busy flag is tested until the not busy state is reached. At that time the reading that was triggered can be read by the iSBC 80/10A. The last statement in this routine jumps to TKDATA which reads the data from the DOU and then executes the return.

```
84
85 ;
86 ; *** SUBROUTINE TO TAKE EXTERNALLY TRIGGERED READING ***
87 ;
88 TRGR:
89        MVI     A,00000010B     ; LD RESET BIT 1 CONTROL WORD
90        OUT     CWR             ; OUTPUT TO 8255#2 CNTL WD REG
91        INR     A               ; MODIFY CONTROL WORD TO SET BIT 1
92        OUT     CWR             ; OUTPUT TO 8255#2 CNTL WD REG
93 TWT:
94        IN      FLG             ; INPUT THE BUSY FLAG
95        ANI     00100000B       ; TEST PORT C BIT 5
96        JNZ     TWT             ; LOOP UNTIL NOT BUSY
97        JMP     TKDATA          ; GO READ DATA FROM DOU AND RETURN
98 ;
99
```

The second method for reading the Fluke 8375 is to rely on the sample rate set from the front panel controls and to wait until a full transition of the busy flag is observed. This guarantees that a previous reading is not mistakenly interpreted as a new reading.

```
100
101 ;
102 ; *** SUBROUTINE TO OBTAIN NEXT READING ***
103 ;
104 NXTRD:
105        IN      FLG             ; INPUT THE BUSY FLAG
106        ANI     00100000B       ; TEST PORT C BIT 5
107        JZ      NXTRD           ; LOOP UNTIL BUSY WITH NEXT READING
108 NXTWT:
109        IN      FLG             ; INPUT THE BUSY FLAG
110        ANI     00100000B       ; TEST PORT C BIT 5
111        JNZ     NXTWT           ; LOOP UNTIL NOT BUSY
112        JMP     TKDATA          ; GO READ DATA FROM DOU AND RETURN
113 ;
114
```

Notice that the loops beginning at NXTWT in the above program segment and at TWT in the previous program segment are identical. This suggests the possibility of some obvious code optimization that is omitted here for the sake of clarity.

There is one subroutine remaining to complete full utilization of the Fluke 8375 DOU capabilities. It is the subroutine to take data from the 8375 DOU. The schematic (Figure 3) shows that port 5 (B) bits PB4–PB0 are used to enable the DOU drivers. Data from the DOU includes:

- 5 decades of digits
- encoded range and overrange

- function: Volts DC, Volts AC, Ohms, Kilohms
- modifiers: Filter, Ext. Ref., Remote
- overload
- trigger

The function of this subroutine is to read five bytes of data from the 8375 DOU and place them in a RAM buffer on the iSBC 80/10A.

```
115 ;
116 ;
117 ; *** SUBROUTINE TO TAKE DATA FROM 8375 DOU ***
118 ;
119 TKDATA:
120         LXI     H,RDBUF     ; LD BUFFER POINTER
121         MVI     A,0EFH      ; SETUP FIRST STROBE
122 TKO:
123         MOV     B,A         ; SAVE CURRENT STROBE
124         OUT     STB         ; STROBE DECADE PAIR
125         IN      DATIN       ; READ DATA
126         MOV     M,A         ; PLACE DATA INTO SBC 80/10 RAM
127         INX     H           ; INCREMENT BUFFER POINTER
128         MOV     A,B         ; RESTORE STROBE
129         RRC                 ; ROTATE TO NEXT STROBE POSITION
130         JC      TKO         ; LOOP UNTIL BIT 0 STROBE DONE
131         OUT     STB         ; DISABLE ALL STROBES
132         RET
133 ;
134
```

This completes the software required to service the Fluke 8375 DOU. The following code consists of a routine to display the data from the interface on the console output device and a short executive program to allow exercising of the driver subroutines.

The display subroutine takes 5 bytes of data from the RAM buffer in which the reading has been stored and prints them, 2 ASCII characters per 8-bit byte, on the console.

```
135 ;
136 ;
137 ; *** SUBROUTINE TO DISPLAY READING BUFFER ON CONSOLE ***
138 ;
139 DISPLAY:
140         LXI     H,RDBUF     ; LD BUFFER POINTER
141         MVI     D,5         ; INITIALIZE COUNTER·
142 DISPO:
143         MOV     A,M         ; LD NEXT BYTE FROM BUFFER
144         CALL    NMOUT       ; CALL SBC 80/10 MONITOR SUBROUTINE
145                             ; TO DISPLAY ACCUMULATOR CONTENTS
146         INX     H           ; INCREMENT BUFFER POINTER
147         DCR     D           ; DECREMENT COUNTER
148         JNZ     DISPO       ; LOOP FOR 5 DISPLAY BYTES
149         RET
150 ;
151
```

Operator Interface. The short executive program provides a tool for the purposes of exercising the 8375 DOU driver subroutines. The executive begins by calling the initialization subroutine and then continues on to prompt the operator with a '>' on the console. At that point the operator may enter one of the following characters, causing the program to execute the specified subroutine:

	SUBR	DESCRIPTION
T	ETRIG	Enable external trigger
I	DTRIG	Disables external trigger
E	EPOS	Enable programmed timeouts
D	DPOS	Disable programmed timeouts
N	NXTRD	Next reading
S	TRGR	Trigger and get a reading
X	DISPLAY	Display reading buffer

After the operator has entered a command character, the program obtains the address of the subroutine to be executed and proceeds to set up a return address on the stack. This technique allows a load program counter instruction (PCHL) to be used to enter the subroutine and a return instruction (RET) to resume execution of the executive.

```
152 ;
153 ;
154 ; *** SIMPLE EXECUTIVE EXERCISE PROGRAM ***
155 ;
156 START:
157         LXI     SP,STACK    ; SETUP STACK POINTER
158         CALL    INIT        ; INITIALIZE THE SBC 80/10 8255#2
159 EXEC:
160         CALL    CROUT       ; EXEC ENTRY POINT - PRINT <CR><LF>
161         MVI     C,'>'       ; C LOADED WITH PROMPT CHARACTER
162         CALL    CO          ; CONSOLE OUTPUT
163         CALL    GETCH       ; GET CMND CHAR, MASK OFF PARITY
164         CALL    CO          ; PRINT THE CHARACTER ON THE CONSOLE
165         MOV     A,C         ; PUT CHARACTER BACK INTO THE ACCUM
166         LXI     B,NCMDS     ; C CONTAINS LOOP AND INDEX COUNT
167         LXI     H,CTAB      ; HL POINTS TO CMND TABLE
168 EXECO:
169         CMP     M           ; COMPARE TABLE ENTRY AND CHARACTER
170         JZ      EXEC1       ; BRANCH IF EQUAL - CMND RECOGNIZED
171         INX     H           ; ELSE, INCREMENT TABLE POINTER
172         DCR     C           ; DECREMENT LOOP COUNT
173         JNZ     EXECO       ; BRANCH IF NOT AT TABLE END
174         JMP     EXEC        ; ELSE, CMND ILLEGAL - IGNORE IT
175 EXEC1:
176         LXI     H,CADR      ; LD ADR OF TABLE OF CMND SUBRS
177         DAD     B           ; ADD WHAT IS LEFT OF LOOP COUNT
178         DAD     B           ; - EACH ENTRY IN CADR IS 2 BYTES
179         MOV     A,M         ; GET LSP OF ADR OF TABLE ENTRY TO A
180         INX     H           ; POINT TO NXT BYTE IN TABLE
181         MOV     H,M         ; GET MSP OF ADR OF TABLE ENTRY TO H
182         MOV     L,A         ; PUT LSP OF ADR OF TABLE ENTRY TO L
183         LXI     D,EXEC      ; SETUP RETURN ADR ON THE STACK
184         PUSH    D
185         PCHL                ; NEXT INSTR COMES FROM CMND SUBR
186 ;
187
```

The command and address tables as well as the reading buffer follow to complete the application software.

```
188 ;
189 ;
190 ;   COMMAND AND ADDRESS TABLES
191 ;
192 CTAB:
193         DB      'XSNDEIT'
194 NCMDS   EQU     $-CTAB      ; NUMBER OF VALID COMMANDS
195 ;
196 CADR:
197         DW      0
198         DW      ETRIG
199         DW      DTRIG
200         DW      EPOS
201         DW      DPOS
202         DW      NXTRD
203         DW      TRGR
204         DW      DISPLAY
205 ;
206 ;   READING BUFFER AND STACK SPACE
207 ;
208 RDBUF:
209         DS      5           ; READING BUFFER
210 ;
211
212
213         END     START       ; TRANSFER ADDRESS IS TO START
```

SUMMARY/CONCLUSIONS

This instrumentation control application has been presented to demonstrate the simple techniques used to apply the iSBC 80/10A to the task of interfacing instrumentation. A natural extension of this example would include the control of the Fluke 8375 RCU, as well as the control of many additional instruments to build a small ATE system.
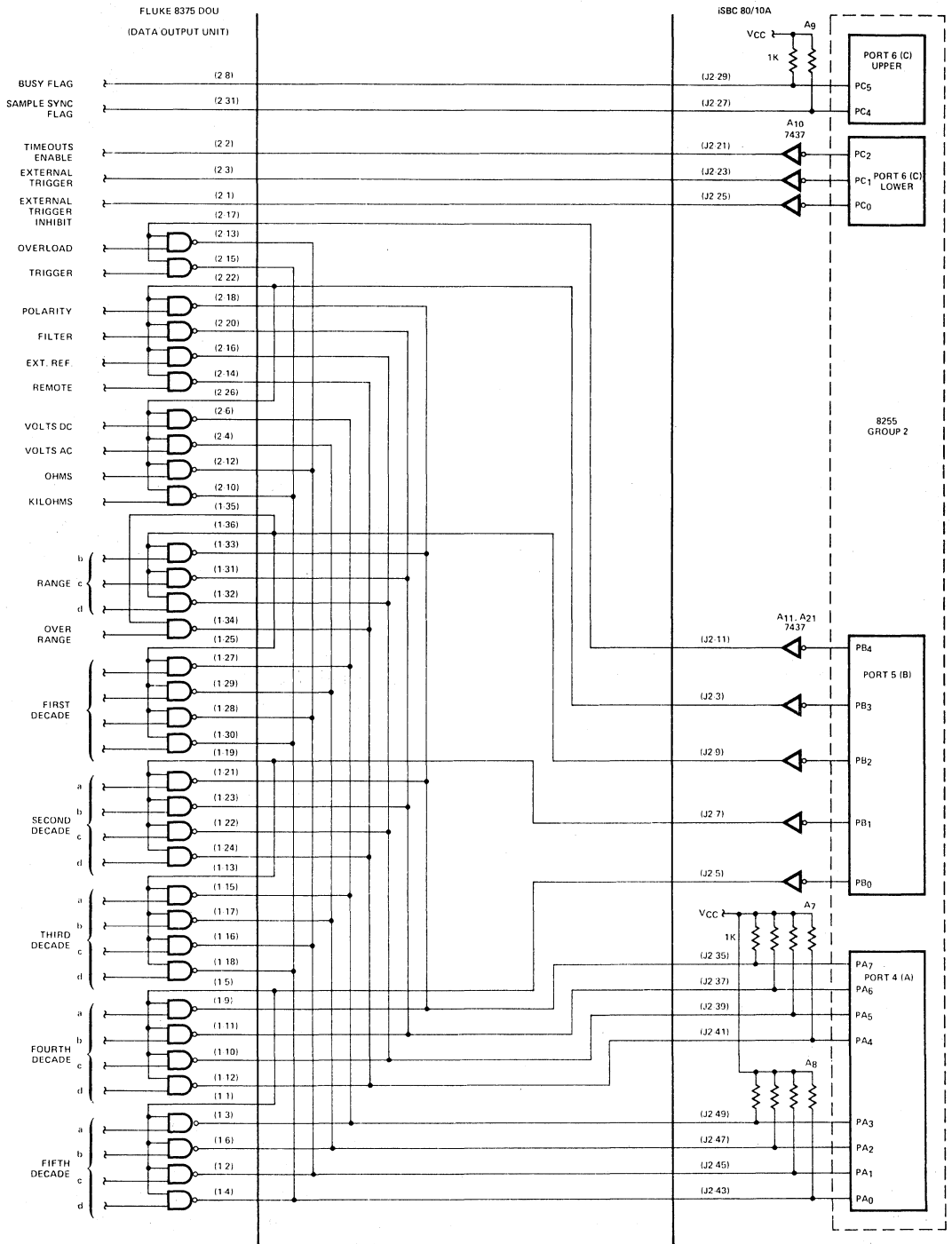
Figure 3. Interface Schematic

COMMUNICATION

A diverse range of single board computer applications exists in the field of communication. The increase in distributed processing generates requirements for self-contained computers to control elements of a communication system, increasing both the throughput and reliability.

There are many situations that necessitate monitoring and controlling a system from a remote site. Typical examples are systems that cover large geographic areas or systems in dangerous environments for human operators. If the object system, which provides the actual parallel inputs and outputs to the plant, is far from the controlling system, you can lower costs by reducing the number of interconnecting wires via the addition of multiplexers to both systems. In the extreme (and often desirable) case of reducing the interconnects to an absolute minimum, all communication between the systems takes place on a single serial data link. If large distances are involved, this link can be standard telephone wires. For moderate distances, the link can be a single twisted pair. In either case, the equipment used to interface the object system to the serial link is called a supervisory control and data acquisition (SCADA) terminal.

The decision to replace a multitude of interconnects with a SCADA terminal is largely economic. Cables and their associated drivers and receivers can represent a significant part of the total cost of a factory automation project, particularly if an electrically noisy environment requires the use of shielded cables. Any potential savings in cabling must, of course, compensate for the additional cost incurred by adding the SCADA terminal to the system.

Communication Application Example

A SCADA terminal demonstrates an industrial communication application of the iSBC 80/10A. The Intel® 8251 USART provides the serial communication link and the two Intel 8255 Programmable Parallel I/O devices provide 48 parallel lines for the object system. A block diagram of a SCADA terminal is shown in Figure 4.

The task of the software in this SCADA terminal example is two-fold. First, it must continually scan its parallel inputs, transmitting the status of those lines in a bit serial mode using the USART. And second, it receives bit serial data from the USART which is to be used to update the parallel outputs. Thus, a major portion of the software deals with
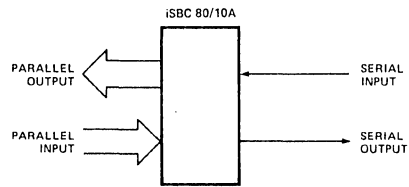
Figure 4. SCADA Terminal Block Diagram

the communications protocol on the serial data lines.

Communications Protocol. A communication protocol is an agreement between communications users that defines the record formats used for data transmissions. The protocol selected for this SCADA terminal application provides the following features:

1. A readable character set to simplify the human interface.

2. Error detection by means of a checksum.

3. Each record specifies the number of data bytes in the record and the initial port number.

Despite its value for human interface, the ASCII character set has problems representing 8-bit binary values, since the high-order bit is not used. Therefore, each binary value is treated as two 4-bit hexadecimal values. Because hexadecimal numbers fall in the range 0–9 and A–F, they can be represented as ASCII characters. However, this representation requires twice as many bytes as a pure binary format.

Record Format. The information encoded into the ASCII hexadecimal format is grouped to form records. Each record has a record mark to flag the beginning of the record, a number of ports specification (record length), destination output start port number, the data field itself, and a checksum.

The record format described below is according to the fields in the record.

Record mark field: Byte 0

The ASCII code for a colon (:) is used to signal the start of a record.

Number of ports field: Byte 1

The number of data bytes in the record is represented by a single ASCII hexadecimal digit in this field. This corresponds to the number of 8-bit

ports to which data will be output by the SCADA terminal in a parallel fashion. The maximum number of data bytes in a record is 15 (F in hexadecimal). A record length of zero is a special case and can be reserved for control information.

Port address field: Byte 2

The single ASCII hexadecimal digit in byte 2 gives the port number of the initial output port. The first data byte is output to the port indicated by the port address; successive bytes are output in successive port locations on the iSBC 80/10A or on expansion I/O boards.

Data field: Bytes 3 to 3+2*(number of ports)−1

An 8-bit binary value is represented by two bytes containing the ASCII characters 0−9 or A−F, which represent a hexadecimal value between 0 and FF (0 and 255 decimal). The high-order digit is in the first byte of each pair.

Checksum field: Bytes 3+2*(number of ports) to 3+2*(number of ports)+1

The checksum field contains the ASCII hexadecimal representation of the two's complement of the 8-bit sum of the 8-bit bytes that result from converting each pair of ASCII hexadecimal digits to one byte of binary, from the number of ports field (the number of ports and port address constitute a pair) to and including the last byte of the data field. Therefore, the sum of all the ASCII pairs in a record after converting to binary, from the number of ports field to and including the checksum field, is zero.

Sample Hexadecimal format:

```
:303A178FF0
  |||    |  |___ Checksum Field
  |||    |_____ Data Field
  |||_____ Starting Port Address
  ||_____ Number of Ports
  |_____ Record Mark
```
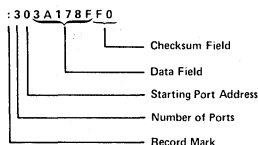
Design Approach Using a State Diagram. Before proceeding to examine the software used to implement the SCADA terminal, consider how the problem might have been approached with TTL logic rather than a microcomputer. The design would likely have been formulated with a state diagram to specify the transitions of a sequential state machine. The sequential-circuit operations would include decoding the serial input records and

encoding the serial output records. An examination of the serial input record state diagram (Figure 5) is useful in understanding some of the procedures encountered later.
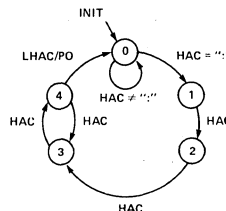
Figure 5. State Diagram

Notes: HAC   = Hexadecimal ASCII character
       LHAC = Last Hexadecimal ASCII character
       PO   = Parallel output

The receipt of an invalid HAC will cause a return to state 0.

The receipt of a colon at any time will produce a transition to state 1.

STATE		DESCRIPTION
0	=	record mark state
1	=	number of ports state
2	=	start port number state
3	=	high-order half of data byte state
4	=	low-order half of data byte state

State 0 is entered at the time of initialization. All state transitions occur when the next character is received. States 1, 2, and 3 are entered with the input of a colon (:), the number of ports and start port number, respectively. States 3 and 4 will cycle as required until all the high and low-order pairs of data have been input. The transition from state 4 to state 0 occurs when the last data byte has been received. If the checksum is correct, the parallel output latches are loaded with the data field of the record.

There are many references to the states contained in this diagram during the discussion of the software procedures. Thus, the state diagram is used as a "flowchart" for the software. As in the other examples in this application note, a textual description accompanies each segment of code. Intel's high-level programming language, PL/M-80, has been used to show the capability to program in a natural, algorithmic language which eliminates the need to manage register usage or memory allocation.

SCADA Terminal Program. The program begins with a comment, that is followed by the program segment label "SCADA". With resident PL/M-80, all programs are considered to be labelled blocks, or modules. This means that all PL/M programs must begin with a LABEL and a DO statement and end with an END statement.

```
        /*
              INDUSTRIAL COMMUNICATION APPLICATION


                      SCADA  TERMINAL
        */

  1       SCADA:
          DO;
```

All variables used in the program must be declared before they can be referred to by their identifiers. This is done by means of a DECLARE statement. In addition to the declaration of variables, macros are declared using the reserved word LITERALLY. These macros are expanded at compile time by textual substitution.

```
  2   1   DECLARE
              SRL$IN$STATE BYTE,
              SRL$IN$PRT BYTE,
              SRL$IN$CNT BYTE,
              PRL$IN$STATE BYTE,
              PRL$IN$STRT$PRT BYTE,
              PRL$IN$NMB$PRTS BYTE,
              SRL$IN$PRL$OUT$BFR(3) BYTE,

              PRL$OUT$PRT$0 LITERALLY '0E5H',
              PRL$OUT$PRT$1 LITERALLY '0EAH',
              PRL$OUT$PRT$2 LITERALLY '0E8H',

              SRL$OUT$STATE BYTE,
              SRL$OUT$PRT BYTE,
              SRL$OUT$CNT BYTE,
              PRL$OUT$STATE BYTE,
              PRL$OUT$STRT$PRT BYTE,
              PRL$OUT$NMB$PRTS BYTE,
              SRL$OUT$PRL$IN$BFR(4) BYTE,

              PRL$IN$PRT$0 LITERALLY '0E4H',
              PRL$IN$PRT$1 LITERALLY '0E6H',
              PRL$IN$PRT$2 LITERALLY '0E9H',

              USART$CMD LITERALLY '0EDH',
              USART$IN LITERALLY '0ECH',
              USART$OUT LITERALLY '0ECH',
              USART$STATUS LITERALLY '0EDH',
              USART$MODE$INSTR LITERALLY '0CFH',
              USART$CMD$INSTR LITERALLY '025H',

              TXRDY LITERALLY '001H',
              RXRDY LITERALLY '002H',

              PPI$CWR$1 LITERALLY '0E7H',
              PPI$CWR$2 LITERALLY '0EBH',
              PPI$CWD$1 LITERALLY '080H',
              PPI$CWD$2 LITERALLY '09BH',

              TRUE LITERALLY '0FFH',
              FALSE LITERALLY '000H',

              FOREVER LITERALLY 'WHILE TRUE',

              NEXT$BYTE BYTE,
              CHECKSUM BYTE;
```

8251 and 8255 Initialization. The INIT procedure sets up the 8251 and 8255's and initializes several variables. Interrupts are disabled to insure that no interrupts are serviced during the execution of the INIT procedure.

```
  3   1   INIT: PROCEDURE;
  4   2       DISABLE;
```

The serial input and serial output state counters are set to state 0. Port number 0 is the parallel input start port and 3 ports of data are input from the parallel ports for serial transmission.

```
  5   2       SRL$IN$STATE = 0;
  6   2       SRL$OUT$STATE = 0;
  7   2       PRL$IN$STRT$PRT = 0;
  8   2       PRL$IN$NMB$PRTS = 3;
```

The Intel 8251 USART must be set up by loading it with mode and command instructions.

The mode instruction format is shown below:
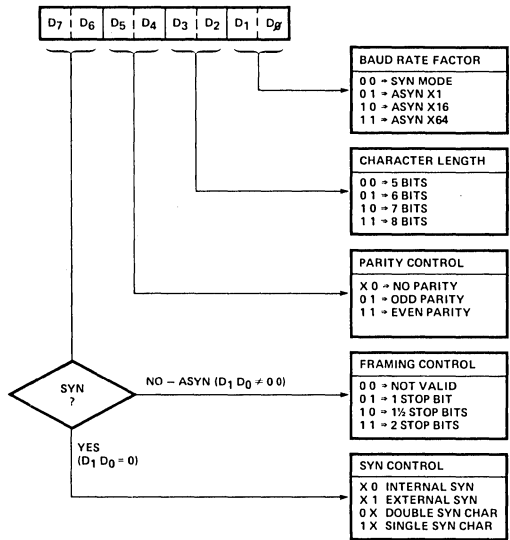
The 8251 characteristics required by this SCADA terminal application include 9600 baud transmission and 8-bit characters. The parallel inputs of the 8255's are periodically scanned. The scanning frequency is determined by the baud rate and number of ports of data being transmitted. For example, the transmission of 3 ports of data requires 11 characters. At a baud rate of 9600 the approximate scan rate is 100 Hz.

The following 8251 mode instruction is used:

Instruction = 1100 1110 Binary = CEH
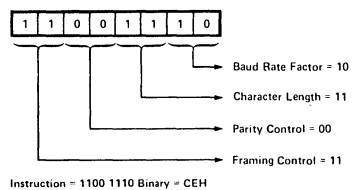
After the mode instruction is sent to the 8251, a command instruction is required to complete the 8251 initialization.

The command instruction format is shown below:
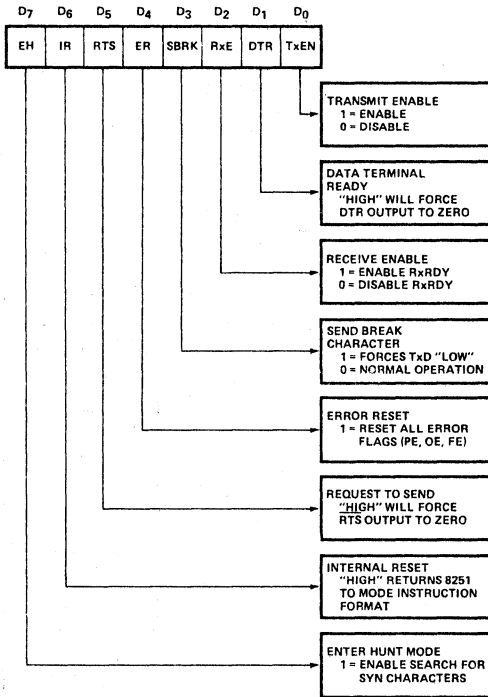
The command instruction enables the transmit and receive functions of the 8251.

The following command instruction is used:
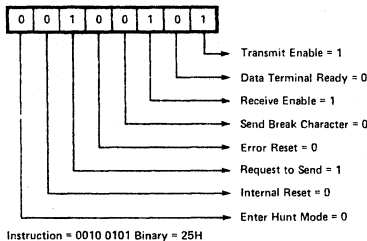
Output instructions send the initialization commands to the 8251. Note that previously declared macros are used to literally replace the mnemonics in the following lines of code.

```
 9    2     OUTPUT(USART$CMD) = USART$MODE$INSTR;
10    2     OUTPUT(USART$CMD) = USART$CMD$INSTR;
```

Initialization of the 8255's is then done to set up the following configurations:

8255 #1

Port 1 (A)	Mode 0	Output
Port 2 (B)	Mode 0	Output
Port 3 (C)	Mode 0	Output

8255 #2

Port 4 (A)	Mode 0	Input
Port 5 (B)	Mode 0	Input
Port 6 (C)	Mode 0	Input

The following command instruction is used for the 8255 #1:
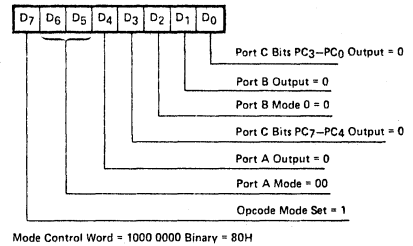
The following command instruction is used for the 8255 #2:
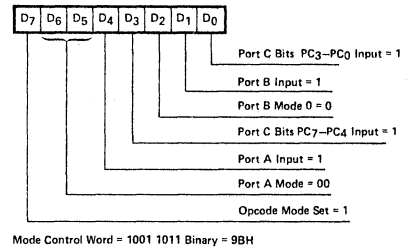
The 8255 initialization commands are given in a similar manner to the 8251 commands.

```
11    2     OUTPUT(PPI$CWR$1) = PPI$CWD$1;
12    2     OUTPUT(PPI$CWR$2) = PPI$CWD$2;
```

The INIT procedure concludes by enabling interrupts.

```
13    2     ENABLE;
14    2     END INIT;
```

Conversion Procedures. Two conversion procedures are required in the program. The first procedure produces a hexadecimal ASCII character from a 4-bit binary value. A typed procedure has been used which returns a value of the type byte. It is called by using its name in an expression.

```
15   1   CHAR$CONV: PROCEDURE (CHAR) BYTE;

16   2      DECLARE CHAR BYTE;

17   2      CHAR = CHAR + '0';
18   2      IF CHAR > '9' THEN
19   2         CHAR = CHAR + 7;
20   2      RETURN CHAR;

21   2   END CHAR$CONV;
```

The second procedure produces a 4-bit binary value from a hexadecimal ASCII character. Because this procedure is used only when a hexadecimal ASCII character is expected, an illegal character (i.e., not a 0–9 or A–F) causes the serial input state counter to indicate state 0. This procedure is also typed. The NMB$CONV procedure emphatically illustrates the point that PL/M-80 performs unsigned arithmetic. Note that when the ASCII value for a zero is subtracted from the digit, NUM = NUM – '0'; a positive number is always produced, even if the value of NUM is less than '0'.

```
22   1   NMB$CONV: PROCEDURE (NMB) BYTE;

23   2      DECLARE NMB BYTE;

24   2      NMB = NMB - '0';
25   2      IF NMB > 9 THEN
26   2      DO;
27   3         IF (NMB > 16) AND (NMB < 23) THEN
28   3            NMB = NMB - 7;
               ELSE
29   3            SRL$IN$STATE = 0;
30   3      END;
31   2      RETURN NMB;

32   2   END NMB$CONV;
```

Parallel Input Procedure. A parallel input procedure is used to input data bytes from the 8255's. The data bytes are then transmitted by the bit serial output device. This procedure also computes the checksum for the serial output record. The checksum, TEMP2, is initialized to contain the parallel input number of ports and the start port, shifted to fit within a single byte. Each cycle of the iterative DO block adds the next data byte to the checksum and places the input data into the SRLOUTPRLINBFR array until the loop is complete. The checksum is then computed as the two's complement of the accumulated sum and also stored in the serial input parallel output buffer.

```
33   1   PARALLEL$IN: PROCEDURE;

34   2      DECLARE (TEMP1,TEMP2) BYTE;

35   2      TEMP2 = PRL$IN$NMB$PRTS * 16 + PRL$IN$STRT$PRT;

36   2      DO PRL$IN$STATE = PRL$IN$STRT$PRT TO
                             PRL$IN$STRT$PRT + PRL$IN$NMB$PRTS - 1;

37   3         DO CASE PRL$IN$STATE;

                  /* PRL IN PRT 0 */
38   4            TEMP1 = INPUT(PRL$IN$PRT$0);

                  /* PRL IN PRT 1 */
39   4            TEMP1 = INPUT(PRL$IN$PRT$1);

                  /* PRL IN PRT 2 */
40   4            TEMP1 = INPUT(PRL$IN$PRT$2);

41   4         END;

42   3         SRL$OUT$PRL$IN$BFR(PRL$IN$STATE) = TEMP1;
43   3         TEMP2 = TEMP2 + TEMP1;

44   3      END;

45   2      SRL$OUT$PRL$IN$BFR(PRL$IN$STRT$PRT + PRL$IN$NMB$PRTS) = -TEMP2;

46   2   END PARALLEL$IN;
```

Parallel Output Procedure. When a complete serial input record has been received and the checksum is correct, the transition from state 4 to state 0 is accompanied by the parallel output of the data from the data field of the serial input record. The parallel output starting port and the number of ports of data is contained in the input record and is thus used in directing the parallel output operation. An iterative DO block increments the PRLOUTSTATE index variable through the required ports and a DO CASE block selectively executes one of the OUTPUT statements for each cycle of the loop.

```
47   1   PARALLEL$OUT: PROCEDURE;

48   2      DECLARE TEMP BYTE;

49   2      DO PRL$OUT$STATE = PRL$OUT$STRT$PRT TO
                              PRL$OUT$STRT$PRT + PRL$OUT$NMB$PRTS - 1;

50   3         TEMP = SRL$IN$PRL$OUT$BFR(PRL$OUT$STATE);

51   3         DO CASE PRL$OUT$STATE;

                  /* PRL OUT PRT 0 */
52   4            OUTPUT(PRL$OUT$PRT$0) = TEMP;

                  /* PRL OUT PRT 1 */
53   4            OUTPUT(PRL$OUT$PRT$1) = TEMP;

                  /* PRL OUT PRT 2 */
54   4            OUTPUT(PRL$OUT$PRT$2) = TEMP;

55   4         END;
56   3      END;

57   2   END PARALLEL$OUT;
```

Serial Input and Output Procedures. The next two procedures contain the software implementations of the state diagram described previously. The processing during each state of the first procedure, the serial character input procedure, is described in the following text.

The procedure begins by reading a character from the 8251 and then converts the character into a 4-bit binary value using the number conversion procedure. The DO CASE block is the mechanism by which a program segment is selected to examine

the input character, provide the required outputs, and to specify the transition to the next state.

```
58   1      SERIAL$CHAR$IN: PROCEDURE;

59   2          DECLARE (CHAR,TEMP) BYTE;

60   2          CHAR = INPUT(USART$IN) AND 07FH;
61   2          TEMP = NMB$CONV(CHAR);

62   2          DO CASE SRL$IN$STATE;
```

State 0 is entered through the initialization process, at the completion of the processing of a serial input record, or when an invalid character has been received. The serial input state will remain 0 until a colon (:) is received, at which time a transition to state 1 is specified.

```
               /* SRL IN STATE 0  =  RECORD MARK */
63   3          DO;
64   4              IF CHAR = ':' THEN
65   4                  SRL$IN$STATE = 1;
66   4          END;
```

The parallel output number of ports is obtained, the counter initialized, and a transition to state 2 is specified from state 1.

```
               /* SRL IN STATE 1  =  NMB PRTS */
67   3          DO;
68   4              PRL$OUT$NMB$PRTS = TEMP;
69   4              SRL$IN$CNT = TEMP;
70   4              SRL$IN$STATE = 2;
71   4          END;
```

In state 2 the parallel output starting port number is obtained, the serial input port is initialized, the checksum is set to contain the parallel output number of ports and starting port, and a transition to state 3 is specified.

```
               /* SRL IN STATE 2  =  STRT PRT */
72   3          DO;
73   4              PRL$OUT$STRT$PRT = TEMP;
74   4              SRL$IN$PRT = TEMP;
75   4              CHECKSUM = PRL$OUT$NMB$PRTS*16 + PRL$OUT$STRT$PRT;
76   4              SRL$IN$STATE = 3;
77   4          END;
```

In state 3 the high-order half of a data byte is obtained and shifted into the proper position of the NEXT$BYTE variable. A transition is specified to state 4.

```
               /* SRL IN STATE 3  =  HI ORDER HALF DATA BYTE */
78   3          DO;
79   4              NEXT$BYTE = TEMP*16;
80   4              SRL$IN$STATE = 4;
81   4          END;
```

State 4 is the final state and requires more processing than the others. First, a whole byte of data is assembled by adding the low and high-order data halves, and then testing to determine if the checksum has been received. If so, and the checksum is correct, the parallel output procedure is executed. Once the entire serial input record has been received, a transition is specified to state 0 whether the checksum is correct or not. However, if the

serial input count has not been exhausted, the assembled byte is placed into the serial input parallel output buffer and a transition back to state 3 is specified.

```
               /* SRL IN STATE 4  =  LO ORDER HALF DATA BYTE */
82   3          DO;
83   4              NEXT$BYTE = NEXT$BYTE + TEMP;
84   4              CHECKSUM = CHECKSUM + NEXT$BYTE;
85   4              IF SRL$IN$CNT = 0 THEN
86   4              DO;
87   5                  IF CHECKSUM = 0 THEN
88   5                      CALL PARALLEL$OUT;
89   5                  SRL$IN$STATE = 0;
90   5              END;
               ELSE
91   4              DO;
92   5                  SRL$IN$PRL$OUT$BFR(SRL$IN$PRT) = NEXT$BYTE;
93   5                  SRL$IN$PRT = SRL$IN$PRT + 1;
94   5                  SRL$IN$CNT = SRL$IN$CNT - 1;
95   5                  SRL$IN$STATE = 3;
96   5              END;
97   4          END;

98   3      END; /* END OF CASES */

99   2  END SERIAL$CHAR$IN;
```

The serial character output procedure is similar to the serial character input procedure. During state 0 the parallel inputs of the 8255's are stored in the serial output parallel input buffer for transmission.

```
100   1      SERIAL$CHAR$OUT: PROCEDURE;

101   2          DECLARE (CHAR,TEMP) BYTE;

102   2          CHAR = 0;

103   2          DO CASE SRL$OUT$STATE;
               /* SRL OUT STATE 0  =  RECORD MARK */
104   3          DO;
105   4              CHAR = ':';
106   4              CALL PARALLEL$IN;
107   4              SRL$OUT$STATE = 1;
108   4          END;

               /* SRL OUT STATE 1  =  NMB PRTS */
109   3          DO;
110   4              TEMP = PRL$IN$NMB$PRTS;
111   4              SRL$OUT$CNT = TEMP;
112   4              SRL$OUT$STATE = 2;
113   4          END;

               /* SRL OUT STATE 2  =  STRT PRT */
114   3          DO;
115   4              TEMP = PRL$IN$STRT$PRT;
116   4              SRL$OUT$PRT = TEMP;
117   4              SRL$OUT$STATE = 3;
118   4          END;

               /* SRL OUT STATE 3  =  HI ORDER HALF DATA BYTE */
119   3          DO;
120   4              TEMP = SHR(SRL$OUT$PRL$IN$BFR(SRL$OUT$PRT),4);
121   4              SRL$OUT$STATE = 4;
122   4          END;

               /* SRL OUT STATE 4  =  LO ORDER HALF DATA BYTE */
123   3          DO;
124   4              TEMP = SRL$OUT$PRL$IN$BFR(SRL$OUT$PRT) AND OFH;
125   4              IF SRL$OUT$CNT = 0 THEN
126   4                  SRL$OUT$STATE = 0;
               ELSE
127   4              DO;
128   5                  SRL$OUT$CNT = SRL$OUT$CNT - 1;
129   5                  SRL$OUT$PRT = SRL$OUT$PRT + 1;
130   5                  SRL$OUT$STATE = 3;
131   5              END;
132   4          END;

133   3      END; /* END OF CASES */

134   2      IF CHAR <> ':' THEN
135   2          CHAR = CHAR$CONV(TEMP);
136   2      OUTPUT(USART$OUT) = CHAR;

137   2  END SERIAL$CHAR$OUT;
```

Interrupt Service Routine. The software in this SCADA terminal application example is interrupt driven. Interrupts, which occur when the transmitter of the 8251 is ready for another character or when the receiver has obtained a serial character, direct the execution of either the serial input

or output character procedures. The following procedure is entered when an interrupt occurs.

```
138   1    USART$INTERRUPT: PROCEDURE INTERRUPT 7;
139   2        DECLARE STATUS BYTE;
140   2        STATUS = INPUT(USART$STATUS);
141   2        IF (STATUS AND TXRDY) = TXRDY THEN
142   2            CALL SERIAL$CHAR$OUT;
143   2        IF (STATUS AND RXRDY) = RXRDY THEN
144   2            CALL SERIAL$CHAR$IN;
145   2    END USART$INTERRUPT;
```

Main Program. The function of the main program is rather simple. It calls the initialization routine and then loops "FOREVER." Notice that the other software is executed only when an interrupt occurs. Rather than loop idly while waiting for an interrupt, the "main program" could take advantage of excess CPU time by processing some other task.

```
           /***********
           MAIN$PROGRAM:
           ***********/
146   1    CALL INIT;
147   1    DO FOREVER;
148   2    END;


149   1    END;
```

SUMMARY/CONCLUSIONS

Further consideration should be given to error checking in the implementation of a SCADA terminal. A checksum has been used in this example which provides some error detection but no correction.

The industrial communication example in this application note has shown a SCADA terminal. Besides providing a convenient forum in which to explore the use of PL/M in an interrupt-driven environment, this application provides a realistic and almost fully-developed tool for the replacement of a multitude of parallel lines. Two such systems can be connected through the serial lines to provide a parallel to parallel transmission scheme as shown in Figure 6.
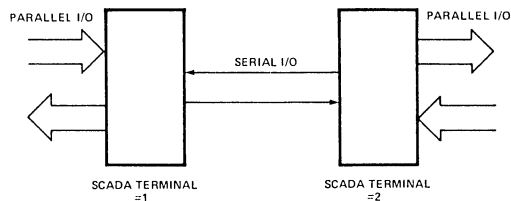
Figure 6. Two SCADA Terminals
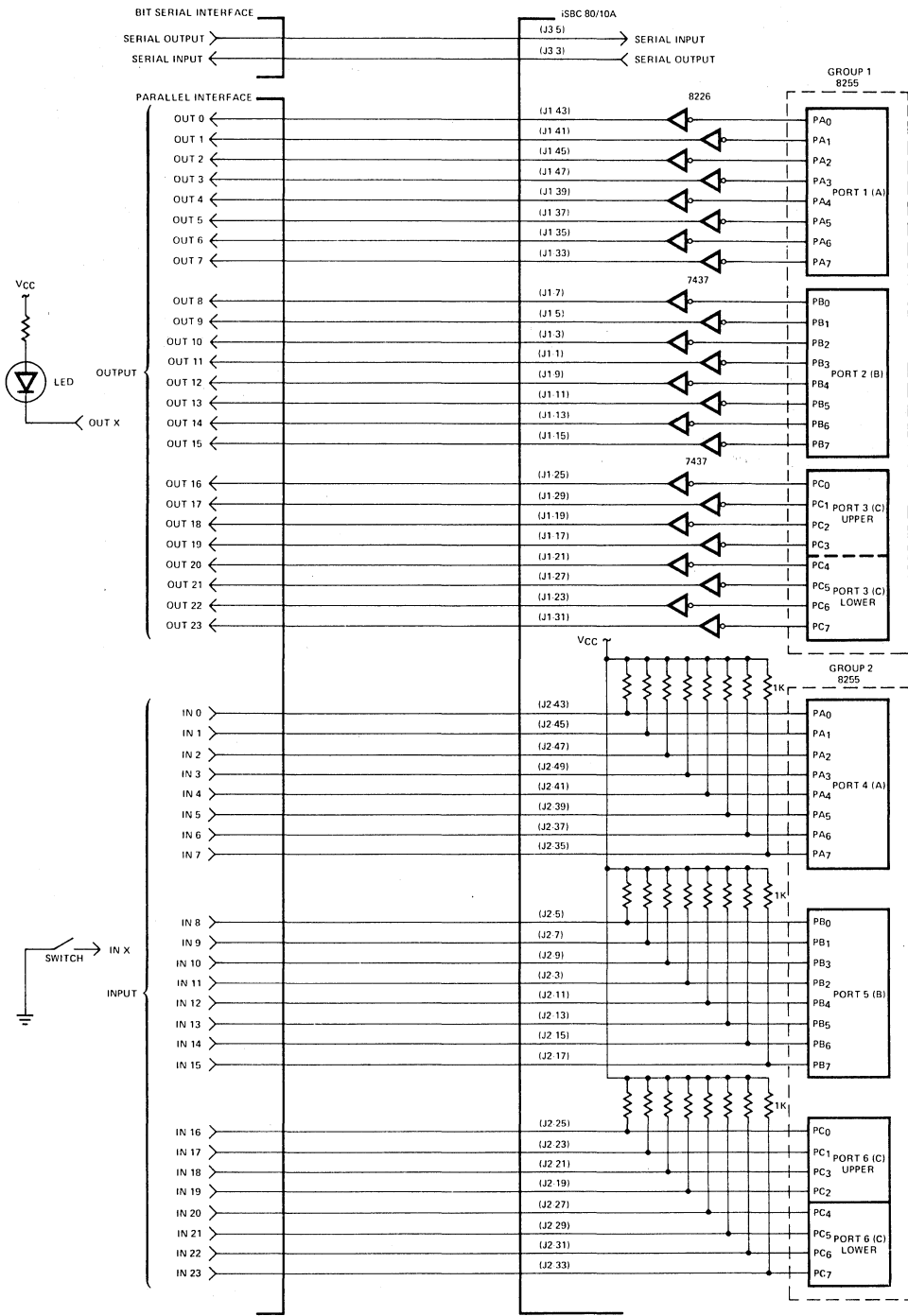
Figure 7. SCADA Terminal Schematic

PROCESS CONTROL

Many single board computers have already been applied in the field of process control. Some of the common denominators observed in these applications include the use of A/D and D/A peripheral boards, process monitoring functions such as servicing display panels for operator interaction, and alarm indicators.

Temperature Monitoring Application Example

A temperature monitoring system has been developed for the purposes of a process control application example. The single open loop system utilizes an A/D converter, a multiplexed display, switches for operator control, and two alarms. A block diagram of the operator's panel is shown in Figure 8 and a schematic in Figure 9.
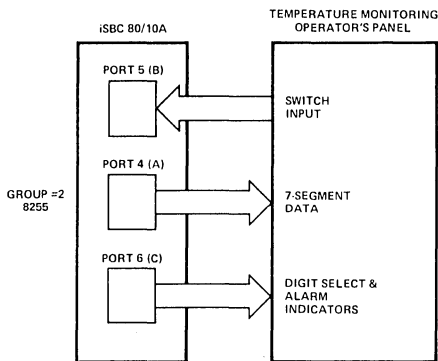
Figure 8. Operator's Panel Block Diagram

Operator's Panel. The operator's panel in this temperature monitoring system contains four 7-segment displays to show the temperature, two light emitting diodes (LEDs) that indicate alarm-low and alarm-high conditions, and six switches. The function of the switches is as follows:

Set Limit — controls whether the current temperature reading is to be displayed (off) or if upper/lower limits are to be set (on).

Set Hi Lo — when set limit is "on", this switch controls whether the low (off) or high (on) limit is to be displayed.

Digit Selects — these two switches control the selection of the digit of the limit which is to be modified. The four binary positions 00, 01, 10 and 11 correspond to the four 7-segment digits.

Leave It — controls whether the digit selected is to be incremented (off) or maintained at its current value (on). When this switch is "off" the digit selected is incremented every 512 ms until the operator turns the switch "on".

Enable Alarm — when set limit is "off" and the current temperature is displayed, this switch controls whether the action of the alarm indicators is to be enabled (on) or disabled (off).

The simple means used to set upper and lower temperature limits is similar to setting the time on a digital wrist watch.

The purpose of the software is to initialize the system and then to enter an endless loop which accumulates 16 readings, updates the displayed reading or handles limit setting, updates the display latches, waits 4 ms, and obtains an A/D reading.

Temperature Monitoring Program. This application example has been coded in Intel's resident PL/M-80 language.

```
        /*
            PROCESS CONTROL APPLICATION

                OPEN LOOP

            TEMPERATURE MONITOR
        */

1       TEMPERATURE$MONITOR:
        DO;
```

The declaration statement includes some dimensioned variables with INITIAL attributes. They provide data strobe positions, a table of bit patterns to convert BCD data to 7-segment data, and a table of the powers of 10 for binary to BCD conversions.

```
2   1     DECLARE
            READING ADDRESS,
            DIGITS(4) BYTE INITIAL (80H,40H,20H,10H),
            BCDTO7SEG(11) BYTE INITIAL (3FH,06H,5BH,4FH,66H,
                            6DH,7CH,07H,7FH,67H,0),
            TENS(4) ADDRESS INITIAL (1000,100,10,1),
            DIGIT$DATA(4) BYTE,
            NXT$DIGIT BYTE,
            UPDATE$COUNT BYTE,
            SET$COUNT BYTE,
            LIMIT(2) ADDRESS,
            ACCUM$RDNG ADDRESS,

            CWR LITERALLY 'OEBH',
            SLCT LITERALLY 'OEAH',
            SEGS LITERALLY 'OE3H',
            SWTS LITERALLY 'OE9H',
            SETUP$PORTS LITERALLY '082H',

            SET$LIMIT LITERALLY '001H',
            SET$HI$LO LITERALLY '002H',
            LEAVE$IT LITERALLY '010H',
            DIGIT$SLCT LITERALLY 'OOCH',
            ENABLE$ALARM LITERALLY '020H',
            SET$ALARM$LO LITERALLY '001H',
            SET$ALARM$HI LITERALLY '003H',
            RESET$ALARM$LO LITERALLY 'OOOH',
            RESET$ALARM$HI LITERALLY '002H',

            TRUE LITERALLY 'OFFH',
            FOREVER LITERALLY 'WHILE TRUE';
```

The analog to digital conversion procedure has been coded in assembly language and is not included in this application note. It is declared as an external typed procedure with no arguments and returns a value of the type address. The value returned is the current temperature. The ATOD procedure is linked later in a step to produce an absolute load module of the entire program.

```
3    1    ATOD: PROCEDURE ADDRESS EXTERNAL;
4    2    END ATOD;
```

Bit set/reset functions of the 8255 have been used to control the alarm-low and high output bits. Use of this function allows individual bits to be controlled without affecting others of port C which are concurrently selecting the digit to be multiplexed on the display.

```
5    1    RESET$ALARMS: PROCEDURE;
6    2        OUTPUT(CWR) = RESET$ALARM$LO;
7    2        OUTPUT(CWR) = RESET$ALARM$HI;
8    2    END RESET$ALARMS;
```

The following procedure is used to initialize the 8255 and several program variables.

```
9    1    INIT: PROCEDURE;
10   2        OUTPUT(CWR) = SETUP$PORTS;
11   2        CALL RESET$ALARMS;
12   2        NXT$DIGIT = 0;
13   2        UPDATE$COUNT = 0;
14   2        SET$COUNT = 7;
15   2        READING = 0;
16   2        ACCUM$RDNG = 0;
17   2        LIMIT(0) = 0000;
18   2        LIMIT(1) = 9999;
19   2    END INIT;
```

A multiplexed display is controlled by the software. Two ports of an 8255 are required for this function shown in Figure 9. The first output port holds the data which drives the four 7-segment displays in parallel. The second output port contains four strobes, each going to a separate common cathode of one of the 7-segment displays.

The update display procedure begins by blanking 7-segment data in the output port. This step avoids shadows that would be produced if the data for the next digit position were loaded prior to updating the strobe. The strobe is then advanced, retaining the alarm bits that occupy other bits of the same output port. Note that an output configured 8255 port can be read with an 8080A INPUT instruction to determine the currently latched output data. The BCD data is obtained from the next digit position of the DIGIT$DATA array and used as a subscript into a table of BCDTO7SEG data. The 7-segment data is also output to the 8255 port in the same statement. The procedure concludes by advancing the NXT$DIGIT pointer.

```
20   1    DISPLAY$UPDATE: PROCEDURE;
21   2        OUTPUT(SEGS) = 0;
22   2        OUTPUT(SLCT) = (DIGITS(NXT$DIGIT) OR (INPUT(SLCT) AND 03H));
23   2        OUTPUT(SEGS) = BCDTO7SEG(DIGIT$DATA(NXT$DIGIT));
24   2        NXT$DIGIT = (NXT$DIGIT+1) AND 03H;
25   2    END DISPLAY$UPDATE;
```

Binary to BCD Conversion. Binary data from the A/D converter must be converted to BCD before it can be used by the DISPLAY$UPDATE procedure to show the current temperature reading. The BINTOBCD procedure performs this conversion operation.

```
26   1    BINTOBCD: PROCEDURE;
27   2        DECLARE (BCD,I) BYTE;
28   2        DO I = 0 TO 3;
29   3            BCD = 0;
30   3            DO WHILE READING >= TENS(I);
31   4                READING = READING - TENS(I);
32   4                BCD = BCD + 1;
33   4            END;
34   3            DIGIT$DATA(I) = BCD;
35   3        END;
36   2    END BINTOBCD;
```

BCD to Binary Conversion. The reverse conversion process is also needed. That is, BCD data must be converted to binary. This procedure is used to take limits, which are set by manipulating BCD digits, and convert them to binary data for use in testing against current temperature readings. Based variables have been used in this procedure to allow access to the actual variables used as arguments in the calling program.

```
37   1    BCDTOBIN: PROCEDURE (BCD$ARRAY$ADR,BIN$DATA$ADR);
38   2        DECLARE
                (BCD$ARRAY$ADR,BIN$DATA$ADR) ADDRESS,
                (BCD$ARRAY BASED BCD$ARRAY$ADR) (4) BYTE,
                (BIN$DATA BASED BIN$DATA$ADR) ADDRESS,
                I BYTE;
39   2        BIN$DATA = 0;
40   2        DO I = 0 TO 3;
                /* BIN$DATA = BIN$DATA*10 + BCD$ARRAY(I) */
41   3            BIN$DATA = SHL(BIN$DATA,1) + SHL(BIN$DATA,3) + BCD$ARRAY(I);
42   3        END;
43   2    END BCDTOBIN;
```

Updating the Display. The UPDATE procedure is entered each time 16 readings have been taken from the A/D converter. The UPDATE$COUNT is reset and the operator switches are input to control the execution path through the procedure. The accumulated reading, which is the total of 16 A/D readings, is divided by 16 to obtain an average reading. Then the accumulated reading is zeroed.

```
44   1      UPDATE: PROCEDURE;

45   2          DECLARE (SWT$FLG,HI$LO,DIGIT) BYTE;

46   2          UPDATE$COUNT = 15;
47   2          SWT$FLG = INPUT(SWTS);
48   2          READING = SHR(ACCUM$RDNG,4);
49   2          ACCUM$RDNG = 0;
```

Setting Limits. If the set limit switch is ON, the limits are to be dealt with instead of testing and displaying the current temperature reading. The alarms are reset during limit setting. The specified limit is converted to BCD and then the Leave-It switch is tested to see if the digit selected is to be incremented or held constant.

```
50   2          IF (SWT$FLG AND SET$LIMIT) = SET$LIMIT THEN
51   2          DO;
52   3              CALL RESET$ALARMS;
53   3              HI$LO = SHR((SWT$FLG AND SET$HI$LO),1);
54   3              READING = LIMIT(HI$LO);
55   3              CALL BINTOBCD;
56   3              IF (SWT$FLG AND LEAVE$IT) <> LEAVE$IT THEN
```

Another counter is used to control digit incrementing. Its purpose is to control the rate at which the selected digit is to be incremented. The major loop in the program has a 4-millisecond delay. Thus, 16 A/D conversions require a period of 64 ms which provides an update frequency of 16 readings per second. This is too fast to accurately select a desired digit which is being incremented. SET$COUNT insures eight update periods (512 ms) between each increment. After the digit has been incremented, the BCD limit value is converted back to binary to set the respective limit. This concludes the action taken when setting limits.

```
57   3          DO;
58   4              IF SET$COUNT = 0 THEN
59   4              DO;
60   5                  SET$COUNT = 7;
61   5                  DIGIT = SHR((SWT$FLG AND DIGIT$SLCT),2);
62   5                  IF DIGIT$DATA(DIGIT) = 9 THEN
63   5                      DIGIT$DATA(DIGIT) = 0;
                         ELSE
64   5                      DIGIT$DATA(DIGIT) = DIGIT$DATA(DIGIT) + 1;
65   5                  CALL BCDTOBIN(.DIGIT$DATA,.LIMIT(HI$LO));
66   5              END;
                     ELSE
67   4                  SET$COUNT = SET$COUNT - 1;
68   4              END;
69   3          END;
```

Testing the Averaged Reading. If the set limit switch is OFF, then the averaged reading is to be tested and displayed. The averaged reading is converted to BCD and then a test is performed to determine whether the reading is to be compared with the upper and lower limits.

```
                 ELSE
70   2          DO;
71   3              CALL BINTOBCD;
72   3              IF (SWT$FLG AND ENABLE$ALARM) = ENABLE$ALARM THEN
```

The reading is compared with both the upper and lower limits if the alarms have been enabled. The results of the tests are used to set and reset the corresponding alarm output bits.

```
73   3              DO;
74   4                  IF HEADING < LIMIT(0) THEN
75   4                      OUTPUT(CWR) = SET$ALARM$LO;
                         ELSE
76   4                      OUTPUT(CWR) = RESET$ALARM$LO;

77   4                  IF HEADING > LIMIT(1) THEN
78   4                      OUTPUT(CWR) = SET$ALARM$HI;
                         ELSE
79   4                      OUTPUT(CWR) = RESET$ALARM$HI;
80   4              END;
```

If the alarms are not enabled, both the alarms are reset to the "off" condition.

```
                     ELSE
81   3                  CALL RESET$ALARMS;
82   3          END;

83   2      END UPDATE;
```

Main Program. The main program is shown below. Its purpose is to initialize the system and then to cycle, continuously executing the code previously described.

```
            /*************

            MAIN$PROGRAM:

            *************/

84   1      CALL INIT;

85   1      DO FOREVER;

86   2          ACCUM$RDNG = ACCUM$RDNG + READING;

87   2          IF UPDATE$COUNT = 0 THEN
88   2              CALL UPDATE;
                 ELSE
89   2              UPDATE$COUNT = UPDATE$COUNT - 1;

90   2          CALL DISPLAY$UPDATE;
91   2          CALL TIME(40);
92   2          READING = ATOD;

93   2      END;

94   1      END;
```

SUMMARY/CONCLUSIONS

The goal of this application example is to demonstrate some of the common functions required for process control systems. Rather than show a small portion of a larger, more complex problem, this example was chosen because it presents a complete solution to a smaller problem. In summary, refreshing a multiplexed display was shown. Conversion procedures for binary to BCD and BCD to binary were used. A simple technique, in terms of hardware requirements, was used to enter lower and upper test values. And, limits testing was done, providing alarm indicators.
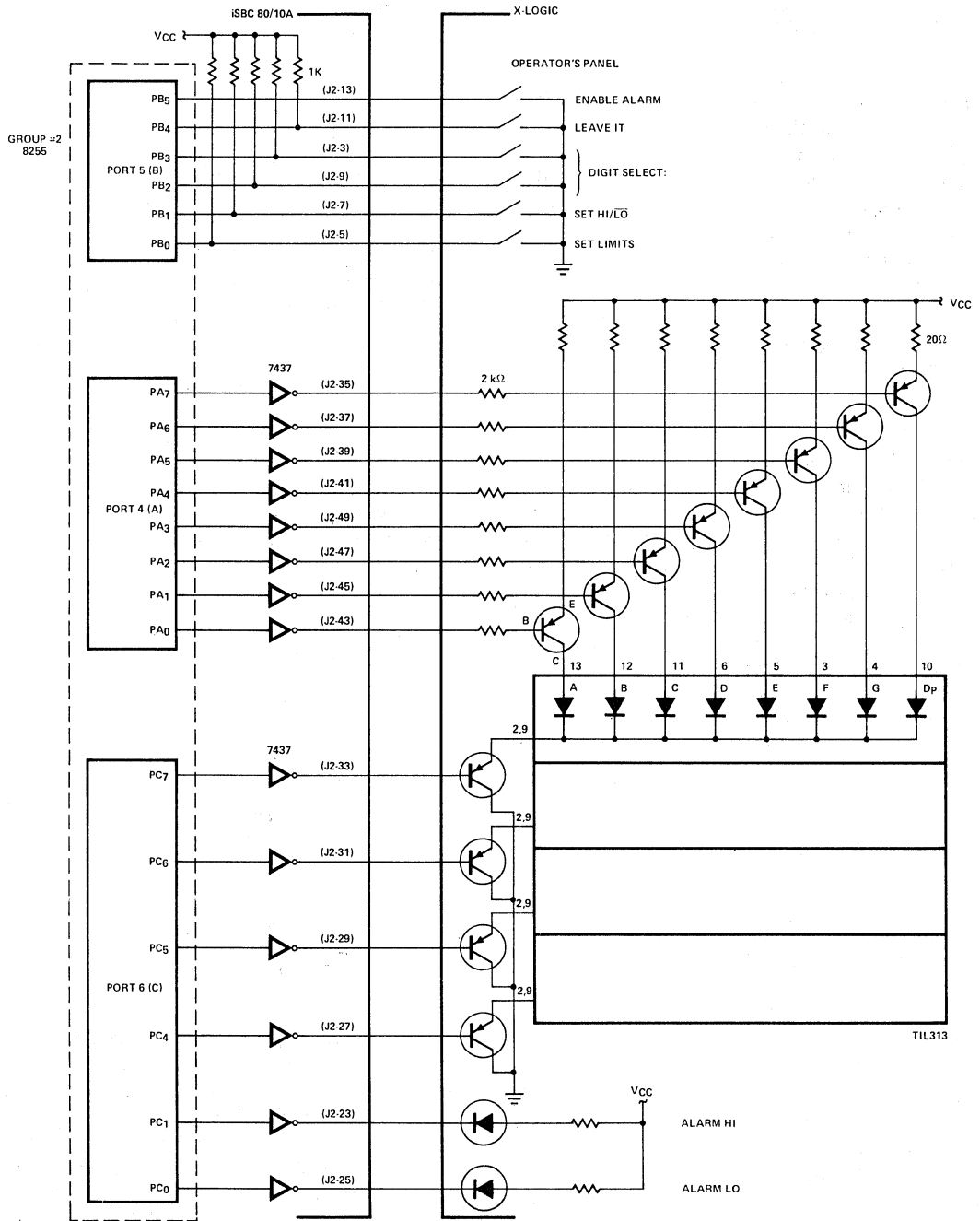
Figure 9. Operator's Panel Schematic

I/O DEVICE CONTROLLER

Peripheral processors have become common elements in computer systems of all sizes and capabilities. The purpose of such a processor is to relieve a central processor from the burden of handling I/O devices. In effect, it is a form of distributed processing. The iSBC 80/10A can be used as a peripheral processor and/or as an I/O device controller. In such a capacity it can significantly reduce the amount of hardware required to interface peripherals. Because the iSBC 80/10A controls only I/O, it is of little consequence that it must do a great deal of detail work that otherwise wastes the processing capability of a larger central processor.

Consider the activity of producing a listing on a line printer. The overhead in maintaining a program in the queue of a central processor which is producing data for a line printer can seriously impact system throughput. If, however, the program were to send the list to a disk file and then command a peripheral processor to take care of the printing, a significant improvement in system performance may be achieved. Printers represent one example of a large number of I/O devices that can be controlled by an iSBC 80/10A. Other devices include cassettes, magnetic tape drives, paper tape readers and punches, etc.

Character Printer Controller Application Example

The control of a Centronics 306 character printer is used as an I/O device controller application example. This example shows the interrupt capability of mode 1 8255 operation. A block diagram of the printer controller is shown in Figure 10 and a schematic in Figure 11.
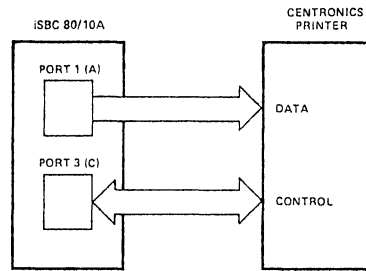
Figure 10. Printer Controller Block Diagram

When the mode 1 or mode 2 configuration is used, software is generally required to support interrupts used in conjunction with handshaking operations. Software routines written for an interrupt driven environment tend to be more complex than status driven routines. The added complexity is because interrupt-driven systems are constructed such that other software tasks are run while the I/O transaction is in progress. A software routine that controls a peripheral device is generally referred to as a device driver. One method of implementing an interrupt-driven device driver is to partition the device driver into a "command processor" and an "interrupt service routine." The command processor is the module that validates and initiates user program requests to the device driver. A common method of passing information between the various software programs is to have the requesting routine provide a device control block in memory. The device control block used in this application example is shown in Table 2.

Table 2. Printer Software Control Block

NAME	POSITION	DEFINITION
Status	Byte 0	A 1-byte field which defines the completion status of an I/O. 00 = Good completion 01 = Error — command already in progress.
Buffer Address	Byte 1, 2	Pointer to the start of the data to print.
Character Count	Byte 3	Count of the number of characters to print.
Character Transferred Count	Byte 4	The number of characters transferred.
Completion Address	Byte 5, 6	Address of a user supplied routine which will be called after the I/O has been performed.

The command processor validates the transaction and initiates the operation described by the control block. Control is then returned to the requester so that other processing may proceed. The interrupt service routine processes the remainder of the transaction.

Interrupt Service Routine Requirements. The interrupt service routine requires the following functions:

1. The state of the machine (registers, status, etc.) must be saved so that it may be restored after the interrupt is processed.

2. The source of the interrupt must be determined. The hardware may support a register which indicates the interrupting device, or the software may poll the device status registers.

3. Data must be passed to or from the device.

4. Control must be passed to the requesting routine at the completion of the I/O.

5. The state of the machine must be restored before returning to the interrupted program.

Printer Controller Program. The software for this application has been coded using Intel® 8080 Macro Assembly Language.

```
0 ;
1 ;*****
2 ;
3 ;         I/O DEVICE CONTROLLER APPLICATION
4 ;
5 ;              INTERRUPT DRIVEN
6 ;
7 ;              CHARACTER PRINTER
8 ;
9 ;*****
```

The following program equates are used to allow symbolic reference to the 8255 ports. Group #1 8255 on the iSBC 80/10A has been used because it will support mode 1 operation.

```
10 ;
11 ;*****
12 ;         PROGRAM EQUATES
13 ;*****
14 PORTA  EQU   0E4H   ; 8255 PORT A
15 PORTB  EQU   0E5H   ; 8255 PORT B
16 PORTC  EQU   0E6H   ; 8255 PORT C
17 CWR    EQU   0E7H   ; 8255 CONTROL WORD REGISTER
```

An initialization control word sent to the control word register of the 8255 will set up the desired configuration.

```
18 ;
19 ;*****
20 ;
21 ;         INITIALIZATION CONTROL WORD
22 ;
23 ;              USED TO CONFIGURE THE 8255 AS FOLLOWS:
24 ;
25 ;                   PORT A - OUTPUT MODE 1
26 ;                   PORT B - INPUT MODE 0 (NOT USED)
27 ;                   PORT C LOWER - OUTPUT
28 ;
29 ;*****
30 ICW    EQU   10101010B      ; INITIALIZATION CONTROL WORD
31 ;*****
```

The bit set/reset capability of the 8255 is used to control the strobe to the printer and to enable/disable interrupts from the 8255.

```
32 ;         SET/RESET CONTROL WORDS
33 ;*****
34 STBON  EQU   00000001B    ; SET STROBE
35 STBOF  EQU   00000000B    ; RESET STROBE
36 ;*****
37 ;         8255 ENABLE/DISABLE INTERRUPT CONTROL WORDS
38 ;*****
39 IEN    EQU   00001101B    ; ENABLE INTERRUPTS
40 IDN    EQU   00001100B    ; DISABLE INTERRUPTS
41 ;*****
```

Device status, control block, and completion equates are shown below.

```
42 ;         DEVICE STATUS EQUATES
43 ;*****
44 LPBSY  EQU   080H   ; BUFFER FULL (LINE PRINTER BUSY)
45 INTRA  EQU   08H    ; INTERRUPT REQUEST
46 ;*****
47 ;         CONTROL BLOCK EQUATES
48 ;*****
49 CBST   EQU   00H    ; STATUS BYTE
50 CBUF   EQU   01H    ; BUFFER ADDRESS
51 CBCC   EQU   03H    ; CHARACTER COUNT
52 CBCT   EQU   04H    ; CHARACTER TRANSFERED COUNT
53 CBCMP  EQU   05H    ; COMPLETION SERVICE ADDRESS
54 ;*****
55 ;         COMPLETION STATUS EQUATES
56 ;*****
57 STGD   EQU   00H    ; GOOD COMPLETION
58 STE1   EQU   01H    ; ERROR - COMMAND ALREADY IN PROGRESS
59 ;*****
```

There are two origin statements in this program. The first origin at 38 hexadecimal is the entry point used when an interrupt is generated by the 8255. A jump instruction to the printer interrupt routine is stored at that location. The second origin at 3000 hexadecimal is the address where the rest of the code will be located.

```
60 ;         RESTART 7 ENTRY POINT
61 ;*****
62         ORG   0038H
63         JMP   PINT
64 ;*****
65 ;         PROGRAM ORIGIN
66 ;*****
67         ORG   3000H
68 ;*****
```

An initialization subroutine issues the mode control word to the 8255 control word register after reset of the device. The printer strobe must then be disabled.

```
69 ;
70 ;         INITIALIZATION ROUTINE
71 ;
72 ;         A,H,L REGISTERS MODIFIED
73 ;
74 ;*****
75 INIT:
76         MVI   A,ICW    ; GET MODE CONTROL WORD
77         OUT   CWR      ; OUTPUT TO CONTROL WORD REGISTER
78         MVI   A,STBON  ; GET SET DATA STROBE CONTROL WORD
79         OUT   CWR      ; SET DATA STROBE (LOW TRUE SIGNAL)
80         RET            ; RETURN TO CALLER
81
```

The command processor is started by the user routine through a subroutine call to PSTRT, with the address of the control block in the D and E registers. The command processor insures that an I/O operation is not already in progress, starts the I/O, enables interrupts, and returns to the caller so that other processing may proceed.

The flowchart and listing for the command processor are shown below.

PSTRT

COMMAND IN PROGRESS? — YES

NO

SET COMMAND IN PROGRESS

ERROR

CLEAR CT

POST TO USER

CALL OUTPUT DATA

ENABLE PROCESSOR INTERRUPTS

RETURN
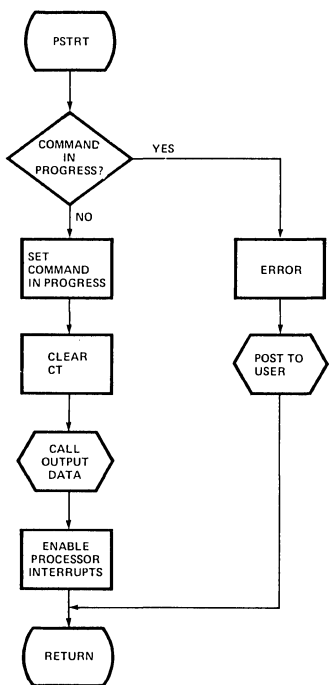
```
82
83 ;*****
84 ;
85 ;      COMMAND PROCESSOR
86 ;
87 ;           INPUTS: CONTROL BLOCK ADDRESS IN D AND E REGISTERS
88 ;
89 ;           OUTPUTS: START I/O OR ERROR STATUS IN CONTROL BLOCK
90 ;
91 ;      A,H,L REGISTERS MODIFIED
92 ;
93 ;*****
94 PSTRT:
95        LDA    PIPRG+1 ; GET PRINT IN PROGRESS BLOCK ADDRESS
96        ANA    A       ; SEE IF ZERO (PRINT IN PROGRESS)
97                       ; IF BLOCK ADDRESS NOT EQUAL TO ZERO THEN
98                       ; PRINT IN PROGRESS
99        JNZ    PSTE    ; IF YES - BRANCH TO ERROR
100       XCHG
101       SHLD   PIPRG   ; SAVE CONTROL BLOCK ADDRESS
102       XCHG
103       LXI    H,CBCT  ; GET INDEX TO CT
104       DAD    D       ; COMPUTE ADDRESS OF CT
105       MVI    M,OOH   ; CLEAR CT
106       CALL   PDATA   ; START I/O
107       EI             ; ENABLE PROCESSOR INTERRUPTS
108       RET            ; RETURN TO CALLER
109 ;*****
110 ; .     ERROR - TRANSACTION ALREADY IN PROGRESS
111 ;*****
112 PSTE:
113       MVI    A,STE1  ; GET ERROR STATUS CODE
114       JMP    POST    ; PASS CONTROL TO COMPLETION ROUTINE
115
```

Interrupt Processing. When the 8255 generates an interrupt, the interrupt request is serviced by the 8080A CPU. The CPU disables processor interrupts and then executes the instruction at location 38 hexadecimal, which is a jump to the interrupt service routine. The interrupt service routine saves the processor state and polls the 8255 to determine the source of the interrupt. Once the interrupting device is identified, the printer output data routine is called. After the entire buffer has been printed, the interrupt service routine passes control to the user-supplied completion routine. Before returning from the interrupt, the state of the processor is restored.

There are a number of error conditions which may occur, such as an interrupt from a device that does not have an active control block, or an interrupt when polling establishes that no device requires service. Neither of these errors should occur, but if they do, the driver should perform in a consistent fashion. The recovery routines implemented to handle these interrupt error conditions are determined by the environment of the particular application.

The flowchart and listing for the printer interrupt service routine are shown below.

INT 7

SAVE REGISTERS

8255? — NO

YES

DISABLE 8255 INT ENABLE PROCESSOR INTERRUPTS

POLL OTHERS & PROCESS

NO — I/O IN PROGRESS?

YES

ERROR

CALL OUTPUT DATA

RESTORE REGISTERS

ENABLE PROCESSOR INTERRUPTS

RETURN
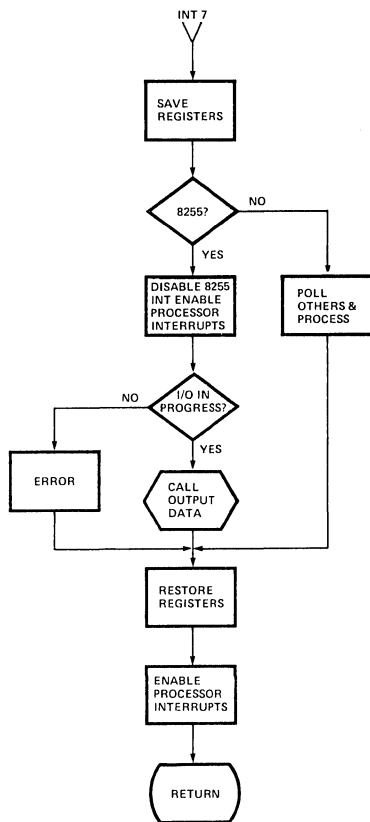
```
116
117 ;*****
118 ;      PRINTER INTERRUPT SERVICE ROUTINE
119 ;      ALL REGISTERS SAVED AND RESTORED
120 ;*****
121 PINT:
122       PUSH   PSW    ; SAVE PSW
123       PUSH   B      ; SAVE REGISTER PAIR B AND C
124       PUSH   D      ; SAVE REGISTER PAIR D AND E
125       PUSH   H      ; SAVE REGISTER PAIR H AND L
126 ;*****
```

```
127 ;     POLL INTERRUPT SOURCE - SEE OF 8255
128 ;*****
129     IN      PORTC   ; GET STATUS OF DEVICE
130     ANI     INTRA   ; SEE IF INT
131     JZ      PPOLL   ; NO -BRANCH TO POLL OTHER DEVICES IF ANY
132     MVI     A,IDN   ; GET 8255 INT DISABLE CONTROL WORD
133     OUT     CWR     ; DISABLE DEVICE INTERRUPTS
134     EI              ; ENABLE PROCESSOR INTERRUPTS
135     LHLD    PIPRG   ; GET CONTROL BLOCK ADDRESS
136     XRA     A       ; CLEAR A REG
137     CMP     H       ; SEE IF PRINT IN PROGRESS
138     JZ      PIER1   ; NO - BRANCH TO ERROR ROUTINE
139     XCHG
140     CALL    PDATA   ; PRINT DATA
141 ;*****
142 ;     RESTORE REGISTERS AND RETURN FROM INTERRUPT
143 ;*****
144 PRTN:
145     POP     H       ; RESTORE REGISTER PAIR H AND L
146     POP     D       ; RESTORE REGISTER PAIR D AND E
147     POP     B       ; RESTORE REGISTER PAIR B AND C
148     POP     PSW     ; RESTORE PSW AND A
149     EI              ; ENABLE PROCESSOR INTERRUPTS
150     RET             ; RETURN TO INTERRUPTED PROCESS
151 ;*****
152 ;     POLL OTHER DEVICES IF ANY
153 ;               IF NO OTHER DIVICES TO POLL - USER SUPPLIED ERROR
154 ;               RECOVERY ROUTINE.
155 ;*****
156 PPOLL:
157     JMP     PRTN    ; RETURN
158 ;*****
159 ;     ERROR - INTERRUPT FROM IDLE DEVICE
160 ;               USER SUPPLIED ERROR RECOVERY ROUTINE
161 ;*****
162 PIER1:
163     JMP     PRTN    ; RETURN
164
```

```
165
166 ;*****
167 ;
168 ;     PRINTER OUTPUT DATA ROUTINE
169 ;
170 ;     CONTROL BLOCK ADDRESS IN D AND E REG
171 ;
172 ;*****
173 PDATA:
174     IN      PORTC   ; GET STATUS OF DEVICE
175     ANI     LPBSY   ; SEE IF BUSY (BUFFER FULL)
176     JZ      PD10    ; IF BUSY - BRANCH
177     LXI     H,CBCT  ; GET INDEX TO CT
178     DAD     D       ; COMPUTER ADDRESS OF CT
179     MOV     A,M     ; GET CT
180     INR     M       ; INC CT
181     DCX     H       ; DEC TO CC
182     CMP     M       ; SEE IF EQUAL
183     JZ      PCOMP   ; IF EQUAL - DONE GO TELL USER
184     LXI     H,CBUF  ; GET INDEX TO BUFFER ADDRESS
185     DAD     D       ; COMPUTE ADDRESS OF BUFFER ADDRESS
186     PUSH    D       ; SAVE D AND E REGISTERS
187     MOV     E,M     ; GET LSB OF BUFFER ADDRESS
188     INX     H       ; INC TO NEXT BYTE
189     MOV     D,M     ; GET BUFFER MSB
190     MVI     H,00H   ; CLEAR H REG
191     MOV     L,A     ; GET CT
192     DAD     D       ; COMPUTER CHARACTER ADDRESS
193     MOV     A,M     ; GET CHARACTER
194     OUT     PORTA   ; OUTPUT CHARACTER TO PRINTER
195     MVI     A,STBOF ; RESET DATA STROBE (LOW TRUE SIGNAL)
196     OUT     CWR
197     INR     A       ; GENERATE SET CONTROL WORD
198     OUT     CWR     ; SET DATA STROBE
199     POP     D       ; RESTORE CONTROL BLOCK ADDRESS
200     JMP     PDATA   ; LOOP UNTIL BUSY
201
```

The printer output data routine places a character in the output buffer of the 8255. Data in the control block is used to direct the transfer of a character. A data strobe signal is then generated through the use of the port C bit set/reset feature.

The flowchart and listing for the printer output data routine are shown below.

If the printer is busy at the time the printer output routine is called, a printer busy routine is executed. The printer busy routine disables the processor interrupts, enables the 8255 interrupts and then enables the processor interrupts on its return to the caller.
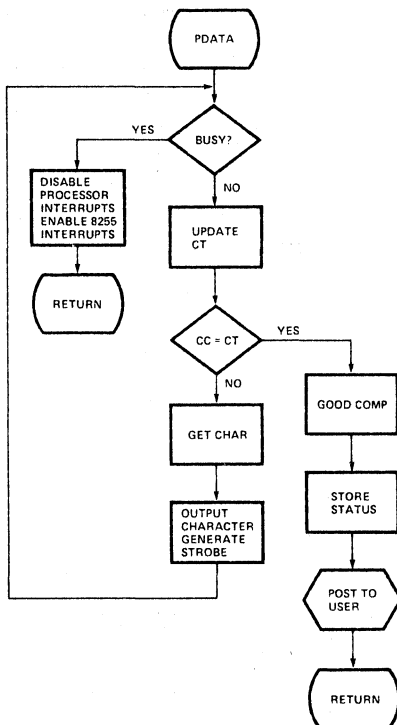

```
202
203 ;*****
204 ;     PRINTER BUSY - RETURN
205 ;*****
206 PD10:
207     DI              ; DISABLE INTERRUPTS
208     MVI     A,IEN   ; ENABLE DEVICE INTERRUPTS
209     OUT     CWR     ; SET INTERRUPT ENABLE
210     RET             ; RETURN TO CALLER
```

When the printer output routine has exhausted the data from the buffer, a good status code is posted to the user. The command in progress flag is also cleared.

```
211 ;*****
212 ;     POST GOOD COMPLETION TO USER
213 ;*****
214 PCOMP:
215     MVI     A,STGD  ; GET GOOD STATUS CODE
216     CALL    POST    ; POST TO USER
217     XRA     A       ; CLEAR A REG
218     STA     PIPRG+1 ; CLEAR COMMAND IN PROGRESS ADDRESS
219     RET             ; RETURN TO CALLER
220
```

The post to user completion routine obtains the completion address from the device control block and passes control to the user routine.

```
221
222 ;*****
223 ;
224 ;     POST TO USER COMPLETION ROUTINE
225 ;
226 ;         INPUTS:  STATUS CODE IN A REG
227 ;                  CONTROL BLOCK ADDRESS IN D AND E REG
228 ;         OUTPUTS: PASSES CONTROL TO USER COMPLETION ADDRES
229 ;                  SPECIFIED IN CONTROL BLOCK
230 ;                  WITH CONTROL BLOCK ADDRESS IN D AND E RE
231 ;
232 ;     A,H,L,B,C REG MODIFIED
233 ;
234 ;*****
```

```
235 POST:
236          XCHG
237          MOV    M,A.     ; UPDATE STATUS
238          XCHG
239          LXI    H,CBCMP  ; GET INDEX TO COMPLETION ADDRESS
240          DAD    D        ; COMPUTE ADDRESS
241          MOV    C,M      ; GET LSB OF COMPLETION ADDRESS
242          INX    H        ; INC TO NEXT BYTE
243          MOV    B,M      ; GET MSB OF COMPLETION ADDRESS
244          PUSH   B        ; PUSH ADDRESS ON STACK
245          RET             ; PASS CONTROL TO USER ROUTINE
246 ;*****
247 ;           DATA AND TABLES
248 ;*****
249          ORG    3D00H
250 PIPRG:   DW     0        ; IN PROGRESS CONTROL BLOCK ADDRESS
251                          ; IF DATA = 0 NO CONTROL BLOCK IN PROGRESS
252                          ; IF DATA <> 0 CONTROL BLOCK IN PROGRESS
253 ;*****
254 ;           END OF MODE ONE EXAMPLE
255 ;*****
256          END
```

SUMMARY/CONCLUSIONS

The iSBC 80/10A has the capability to function in the capacity of a peripheral processor and/or a device controller. This capability is provided in part by the interrupt support logic accompanying the parallel I/O ports. This application example shows how the iSBC 80/10A requires only an interconnect to the device to be controlled.
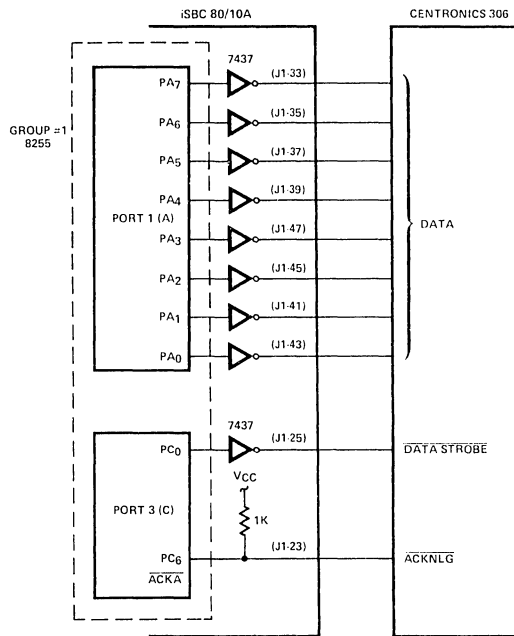
Figure 11. Printer Controller Schematic

CONCLUSION

The purpose of this application note has been to expose the reader to a broad spectrum of potential applications of the Intel iSBC 80/10A and System 80/10 products. Applications have been presented in the areas of instrumentation, communication, process control and I/O device control. The examples were limited to short problems that could be completely described.

Intel's PL/M-80 and 8080 Macro Assembly Language were both used in the examples. Instead of using only assembly language, it was felt that PL/M-80 should also be shown. Coding in an algorithmic language is generally more natural than assembly language and provides these added benefits: reduced program development time and cost, improved product reliability, and easier program maintenance.

While the task of actually configuring the SBC 80/10 for the applications has not been described in this note, detailed instructions are contained in the tables of Chapter 4 in the *iSBC 80/10 and iSBC 80/10A Single Board Computer Hardware Reference Manual.*

The Intel iSBC 80/10A has been designed to provide users with subsystems that have processing power, memory storage, parallel and serial programmable I/O interfaces. A design goal of the iSBC 80/10A was to minimize the requirements for customized interface hardware in user applications. This application note has demonstrated the achievement of this goal. The net effect is to reduce the number of tedious design tasks, thus allowing the systems designer to concentrate on systems integration and other problems unique to his job.

APPENDIX A
iSBC 80/10A SCHEMATICS
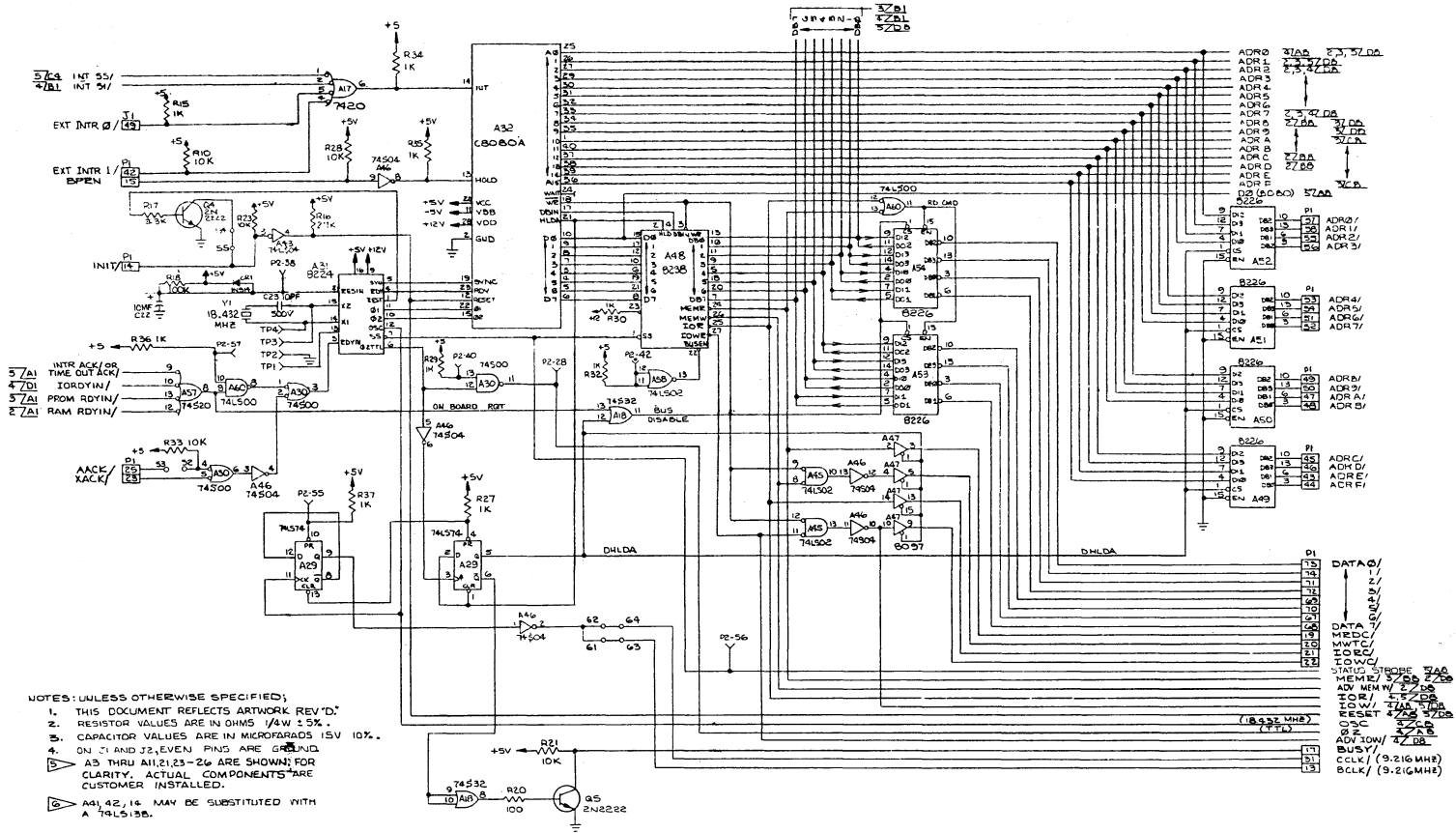
NOTES: UNLESS OTHERWISE SPECIFIED;
1. THIS DOCUMENT REFLECTS ARTWORK REV "D".
2. RESISTOR VALUES ARE IN OHMS 1/4W ±5%.
3. CAPACITOR VALUES ARE IN MICROFARADS 15V 10%.
4. ON J1 AND J2, EVEN PINS ARE GROUND.
5. A3 THRU A11,21,23-26 ARE SHOWN; FOR CLARITY. ACTUAL COMPONENTS ARE CUSTOMER INSTALLED.
6. A41,42,14 MAY BE SUBSTITUTED WITH A 74LS138.

INFORMATION AND SCHEMATICS SUBJECT TO CHANGE WITHOUT NOTICE, FOR REFERENCE ONLY.

INFORMATION AND SCHEMATICS
SUBJECT TO CHANGE WITHOUT
NOTICE, FOR REFERENCE ONLY.

INFORMATION AND SCHEMATICS
SUBJECT TO CHANGE WITHOUT
NOTICE, FOR REFERENCE ONLY.

INFORMATION AND SCHEMATICS
SUBJECT TO CHANGE WITHOUT
NOTICE, FOR REFERENCE ONLY.

INFORMATION AND SCHEMATICS
SUBJECT TO CHANGE WITHOUT
NOTICE, FOR REFERENCE ONLY.

NOTES: UNLESS OTHERWISE SPECIFIED:
1. THIS DOCUMENT REFLECTS ARTWORK "REVA".
2. RESISTOR VALUES ARE IN OHMS, 1/4 W, ±5%.
3. CAPACITOR VALUES ARE IN MICROFARADS, 25V, +80 -20%.
4. ON J1 AND J2 EVEN PINS ARE GROUND.
5. A3 THRU A11, 21, 23 THRU 26 ARE SHOWN FOR CLARITY. ACTUAL COMPONENTS ARE CUSTOMER INSTALLED.
6. A41, 42 & 14 MAY BE SUBSTITUTED WITH A 74LS138.

INFORMATION AND SCHEMATICS
SUBJECT TO CHANGE WITHOUT
NOTICE, FOR REFERENCE ONLY.

INFORMATION AND SCHEMATICS
SUBJECT TO CHANGE WITHOUT
NOTICE, FOR REFERENCE ONLY.

1-41

PROM Ø PROM 1 PROM 2 PROM 3

ADR Ø
1
2
3
4
5
6
7
8
ADR 9

17D1

ADR A
B
C
D
F
ADR E

17D1

17C8 RAM RDY IN/

17B1 MEMR/

OU BOARD
MEM CMD/

DBØ
1
2
3
4
5
DB7

17D4

PROM RDY IN/
17C8

DM Ø
1
2
3
4
5
6
7
DM 7

27B1
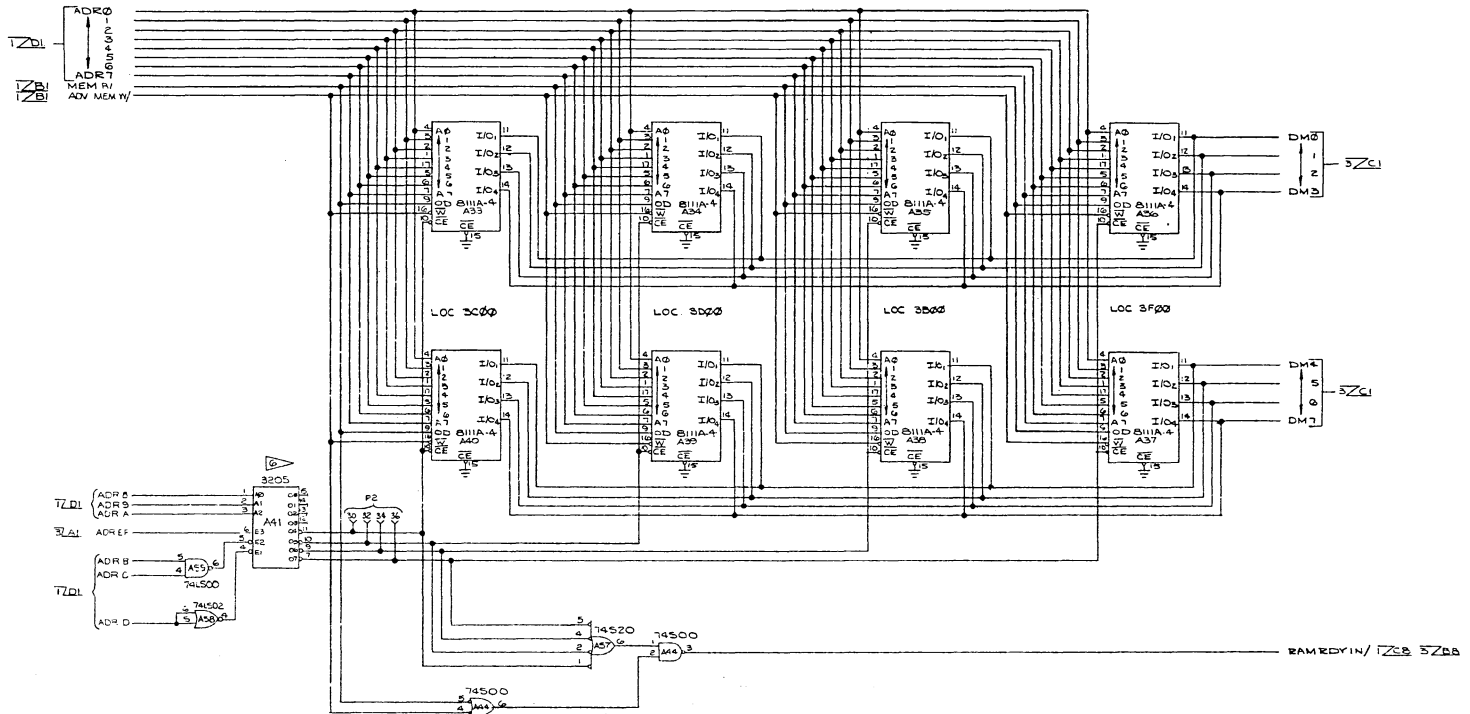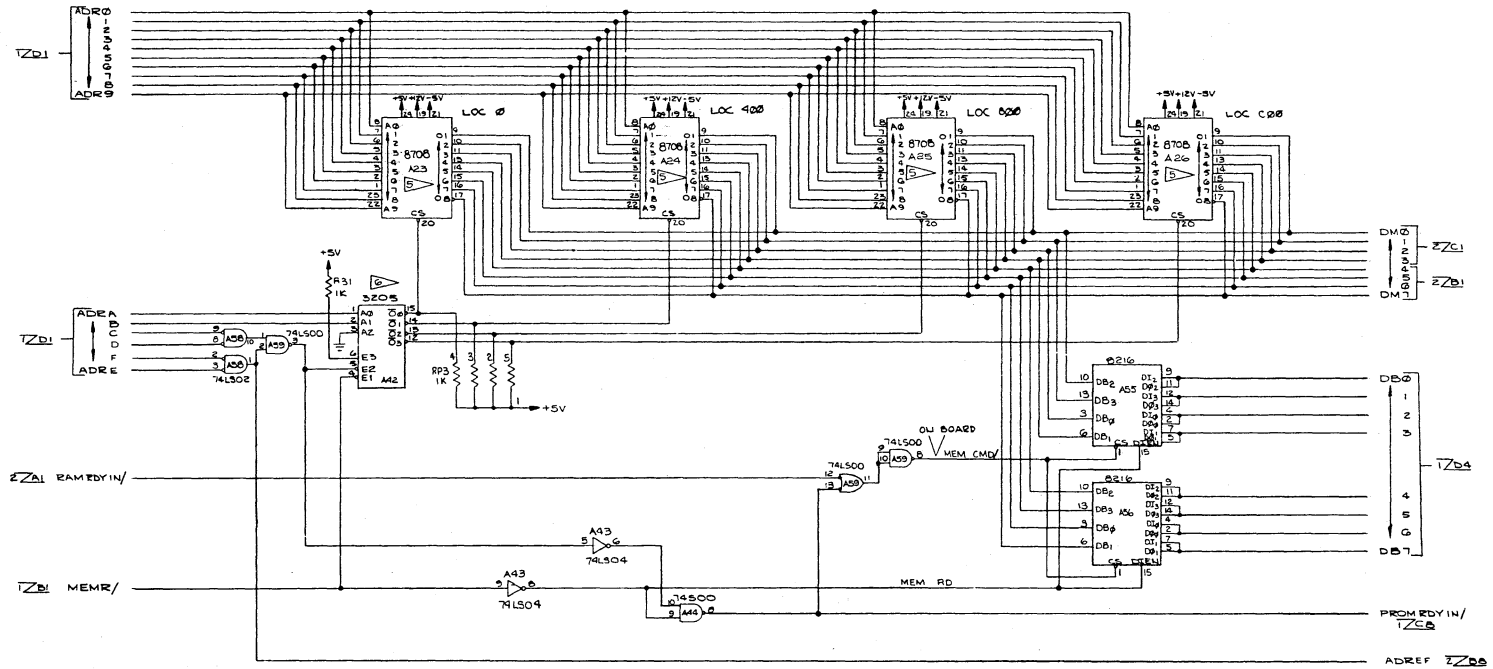
JUMPER TABLE

2716	66-67	69-70	74-75	77-78
2708	65-66	68-69	73-74	76-78

	A23	A24	A25	A26
2716	0-7FF	1000-17FF	800-FFF	1800-1FFF
2708	0-3FF	400-7FF	800-BFF	C00-FFF

A23 8708
A24 8708
A25 8708
A26 8708

A42 3205
A58 74LS00
A59 74LS02
A58 74LS02
A43 74LS04
A44 74300
A59 74LS00
74LS00
A55 8216
A56 8216

R31 1K
RP3 1K
R38 12K
+5V +12V -5V

INFORMATION AND SCHEMATICS
SUBJECT TO CHANGE WITHOUT
NOTICE, FOR REFERENCE ONLY.

INFORMATION AND SCHEMATICS
SUBJECT TO CHANGE WITHOUT
NOTICE, FOR REFERENCE ONLY.

Intel® MULTIBUS™ Interfacing

Joe Barthmaier
OEM Microcomputer
Systems Applications

9800587B

Intel® MULTIBUS™ Interfacing

Contents

I. INTRODUCTION

A significant measure of the power and flexibility of the Intel OEM Computer Product Line can be attributed to the design of the Intel MULTIBUS system bus. The bus structure provides a common element for communication between a wide variety of system modules which include: Single Board Computers, memory, digital, and analog I/O expansion boards, and peripheral controllers.

The purpose of this application note is to help you develop a working knowledge of the Intel MULTI-BUS specification. This knowledge is essential for configuring a system containing multiple modules. Another purpose is to provide you with the information necessary to design a bus interface for a slave module. One of the tools that will be used to achieve this goal is the complete description of a MULTIBUS slave design example. Other portions of this application note provide an in depth examination of the bus signals, operating characteristics, and bus interface circuits.

This application note was originally written in 1977. Since 1977, the MULTIBUS specification has been significantly expanded to cover operation with both 8 and 16-bit system modules and with an auxiliary power bus. This application note now contains information on these new MULTIBUS specification features.

In addition, a detailed MULTIBUS specification has also been published which provides the user with further information concerning MULTIBUS interfacing. The MULTIBUS specification and other useful documents are listed in the overleaf of this note under Related Intel Publications.

II. MULTIBUS™ SYSTEM BUS DESCRIPTION

Overview

The Intel MULTIBUS signal lines can be grouped in the following categories: 20 address lines, 16 bidirectional data lines, 8 multilevel interrupt lines, and several bus control, timing and power supply lines. The address and data lines are driven by three-state devices, while the interrupt and some other control lines are open-collector driven.

Modules that use the MULTIBUS system bus have a master-slave relationship. A bus master module can drive the command and address lines: it can control the bus. A Single Board Computer is an example of a bus master. A bus slave cannot control the bus. Memory and I/O expansion boards are examples of bus slaves. The MULTI-BUS architecture provides for both 8 and 16-bit bus masters and slaves.

Notice that a system may have a number of bus masters. Bus arbitration results when more than one master requests control of the bus at the same time. A bus clock is usually provided by one of the bus masters and may be derived independently from the processor clock. The bus clock provides a timing reference for resolving bus contention among multiple req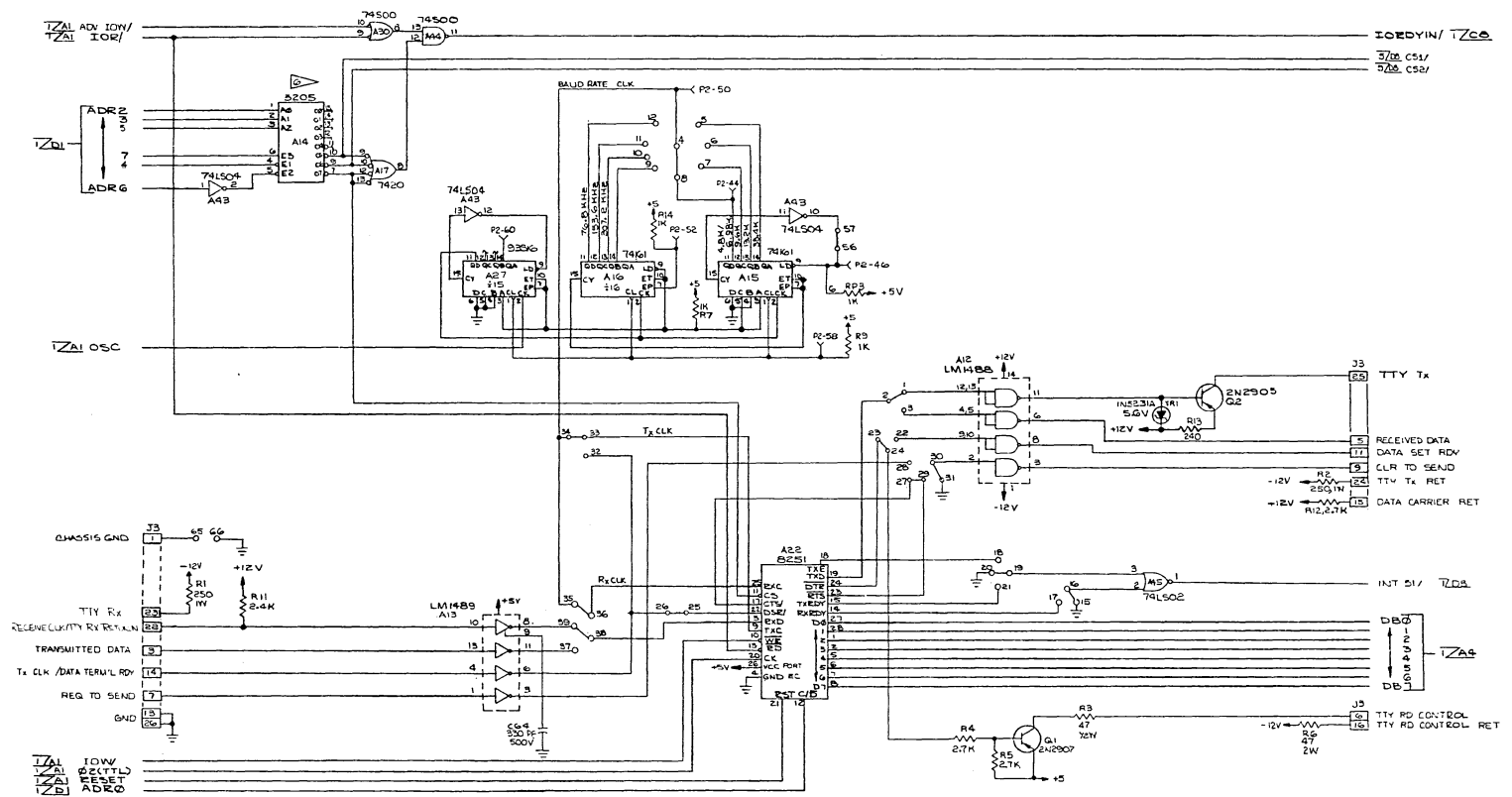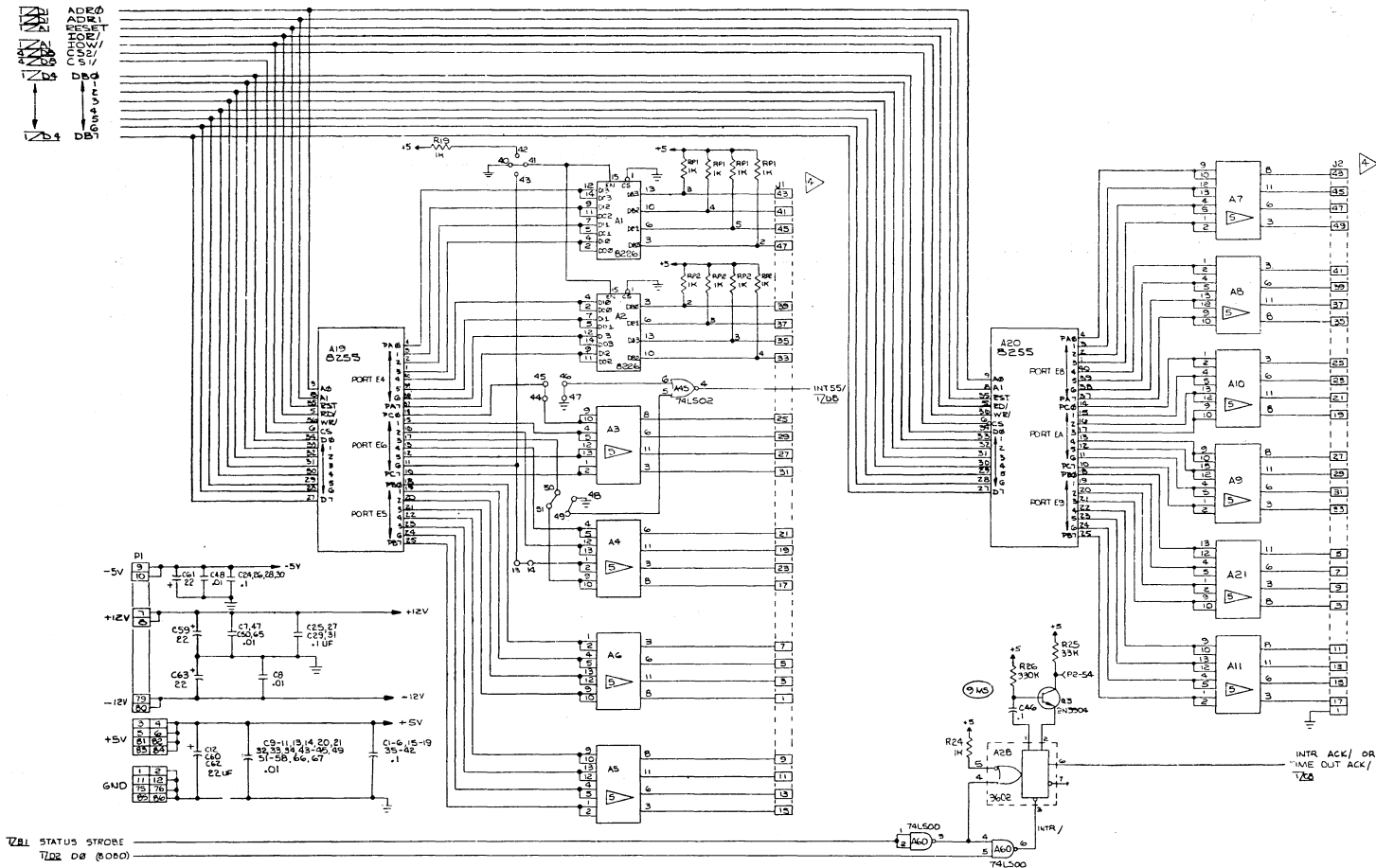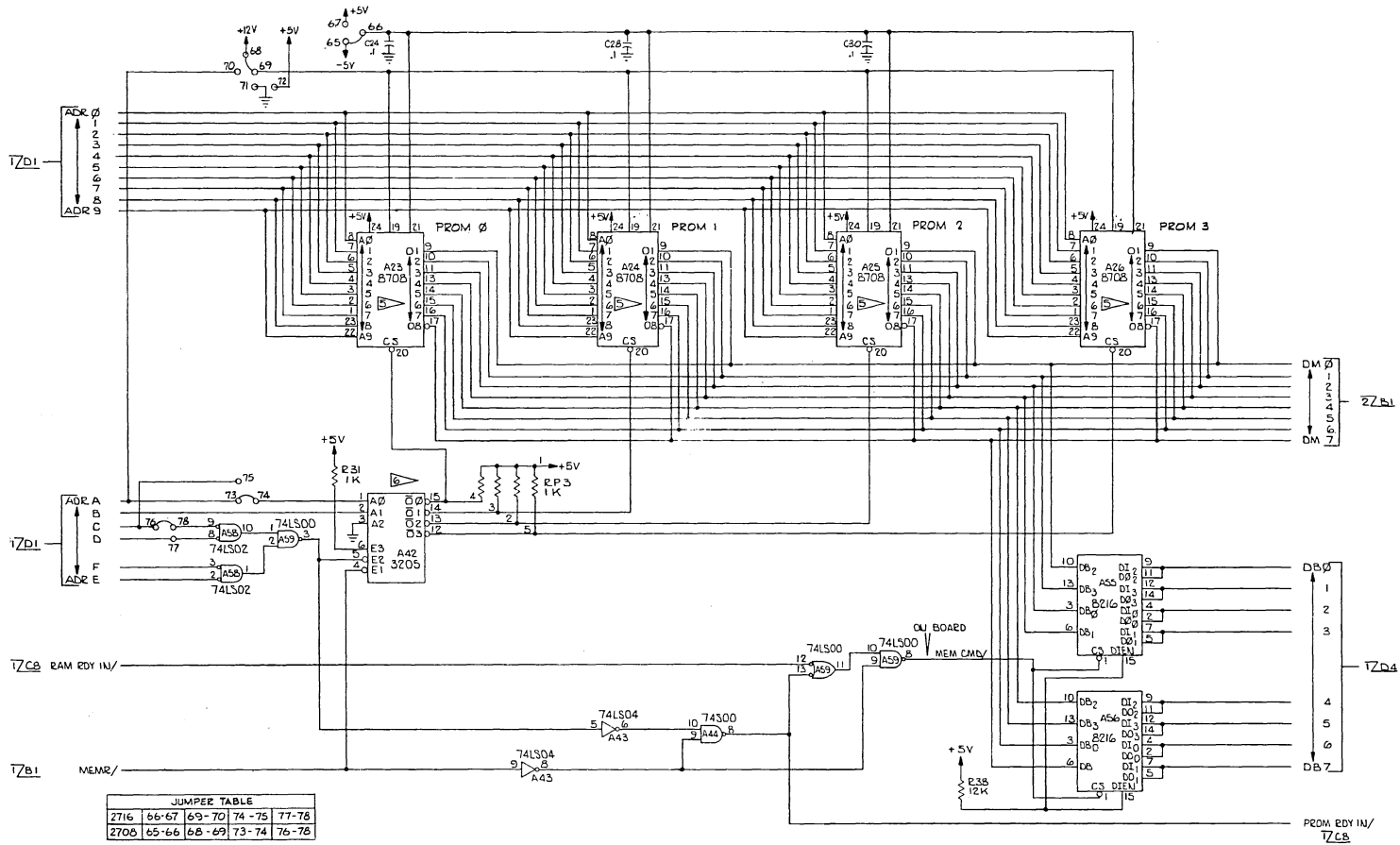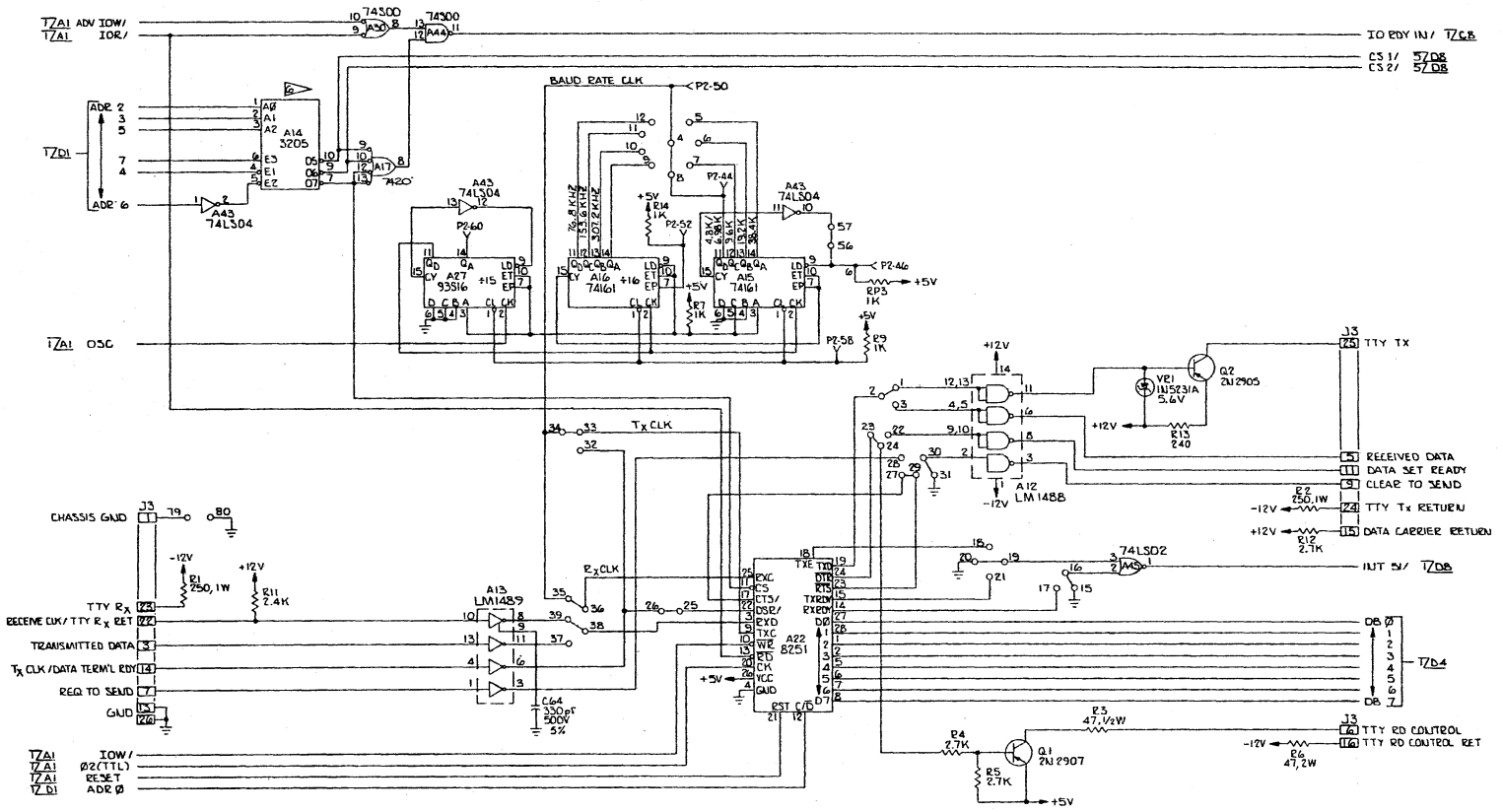uests from bus masters. For example, a processor and a DMA (direct memory access) module may both request control of the bus. This feature allows different speed masters to share resources on the same bus. Actual transfers via the bus, however, proceed asynchronously with respect to the bus clock. Thus, the transfer speed is dependent on the transmitting and receiving devices only. The bus design prevents slow master modules from being handicapped in their attempts to gain control of the bus, but does not restrict the speed at which faster modules can transfer data via the same bus. Once a bus request is granted, single or multiple read/write transfers can proceed. The most obvious applications for the master-slave capabilities of the bus are multi-processor configurations and high-speed direct-memory-access (DMA) operations. However, the master-slave capabilities of the bus are by no means limited to these two applications.

MULTIBUS™ Signal Descriptions

This section defines the signal lines that comprise the Intel MULTIBUS system bus. These signals are contained on either the P1 or P2 connector of boards compatible with the MULTIBUS specification. The P1 signal lines contain the address, data, bus control, bus exchange, interrupt and power supply lines. The P2 signal lines contain the optional auxiliary signal lines. Most signals on the bus are active-low. For example, a low level on a control signal on the bus indicates active, while a low level on an address or data signal on the bus represents logic "1" value.

NOTE

In this application note, a signal will be designated active-low by placing a slash (/) after the mnemonic for the signal.

Appendix A contains a pin assignment list of the following signals:

MULTIBUS P1 Signal Lines —

Initialization Signal Line

INIT/

Initialization signal; resets the entire system to a known internal state. INIT/ may be driven by one of the bus masters or by an external source such as a front panel reset switch.

Address and Inhibit Lines

ADR0/ - ADR13/

20 address lines; used to transmit the address of the memory location or I/O port to be accessed. The lines are labeled ADR0/ through ADR9/, ADRA/ through ADRF/ and ADR10/ through ADR13/. ADR13/ is the most significant bit. 8-bit masters use 16 address lines (ADR0/ - ADRF/) for memory addressing and 8 address lines (ADR0/ - ADR7/) for I/O port selection. 16-bit masters use all twenty address lines for memory addressing and 12 address lines (ADR0/ - ADRB/) for I/O port selection. Thus, 8-bit masters may address 64K bytes of memory and 256 I/O devices while 16-bit masters may address 1 megabyte of memory and 4096 I/O devices. (The 8086 CPU actually permits 16 address bits to be used to specify I/O devices, the MULTIBUS specification, however, states that only the low order 12 address bits can be used to specify I/O ports.) In a 16-bit system, the ADR0/ line is used to indicate whether a low (even) byte or a high (odd) byte of memory or I/O space is being accessed in a word oriented memory or I/O device.

BHEN/

Byte High Enable; the address control line which is used to specify that data will be transferred on the high byte (DAT8/ - DATF/) of the MULTIBUS data lines. With current iSBC boards, this signal effectively specifies that a word (two byte) transfer is to be performed. This signal is used only in systems which incorporate sixteen bit memory or I/O modules.

INH1/

Inhibit RAM signal; prevents RAM memory devices from responding to the memory address on the system address bus. INH1/ effectively allows ROM memory devices to override RAM devices when ROM and RAM memory are assigned the same memory addresses. INH1/ may also be used to allow memory mapped I/O devices to override RAM memory.

INH2/

Inhibit ROM signal; prevents ROM memory devices from responding to the memory address on the system address bus. INH2/ effectively allows auxiliary ROM (e.g., a bootstrap program) to override ROM devices when ROM and auxiliary ROM memory are assigned the same memory addresses. INH2/ may also be used to allow memory mapped I/O devices to override ROM memory.

Data Lines

DAT0/ - DATF/

16 bidirectional data lines; used to transmit or receive information to or from a memory location or I/O port. DATF/ being the most significant bit. In 8-bit systems, only lines DAT0/ - DAT7/ are used (DAT7/ being the most significant bit). In 16-bit systems, either 8 or 16 lines may be used for data transmission.

Bus Priority Resolution Lines

BCLK/

Bus clock; the negative edge (high to low) of BCLK/ is used to synchronize bus priority resolution circuits. BCLK/ is asynchronous to the CPU clock. It has a 100 ns minimum period and a 35% to 65% duty cycle. BCLK/ may be slowed, stopped, or single stepped for debugging.

CCLK/

Constant clock; a bus signal which provides a clock signal of constant frequency for unspecified general use by modules on the system bus. CCLK/ has a minimum period of 100 ns and a 35% to 65% duty cycle.

BPRN/

Bus priority in signal; indicates to a particular master module that no higher priority module is requesting use of the system bus. BPRN/ is synchronized with BCLK/. This signal is not bused on the backplane.

BPRO/

Bus priority out signal; used with serial (daisy chain) bus priority resolution schemes. BPRO/ is passed to the BPRN/ input of the master module with the next lower bus priority. BPRO/ is synchronized with BCLK/. This signal is not bused on the backplane.

BUSY/

Bus busy signal; an open collector line driven by the bus master currently in control to indicate that the bus is currently in use. BUSY/ prevents all other master modules from gaining control of the bus. BUSY/ is synchronized with BCLK/.

BREQ/

Bus request signal; used with a parallel bus priority network to indicate that a particular master module requires use of the bus for one or more data transfers. BREQ/ is synchronized with BCLK/. This signal is not bused on the backplane.

CBRQ/

Common bus request; an open-collector line which is driven by all potential bus masters and is used to inform the current bus master that another master wishes to use the bus. If CBRQ/ is high, it indicates to the bus master that no other master is requesting the bus, and therefore, the present bus master can retain the bus. This saves the bus exchange overhead for the current master.

Information Transfer Protocol Lines

A bus master provides separate read/write command signals for memory and I/O devices: MRDC/, MWTC/, IORC/ and IOWC/, as explained below. When a read/write command is active, the address signals must be stabilized at all slaves on the bus. For this reason, the protocol requires that a bus master must issue address signals (and data signals for a write operation) at least 50 ns ahead of issuing a read/write command to the bus, initiating the data transfer. The bus master must keep address signals unchanged until at least 50 ns after the read/write command is turned off, terminating the data transfer.

A bus slave must provide an acknowledge signal to the bus master in response to a read or write command signal.

MRDC/

Memory read command; indicates that the address of a memory location has been placed on the system address lines and specifies that the contents (8 or 16 bits) of the addressed location are to be read and placed on the system data bus. MRDC/ is asynchronous with respect to BCLK/.

MWTC/

Memory write command; indicates that the address of a memory location has been placed on the system address lines and that data (8 or 16 bits) has been placed on the system data bus. MWTC/ specifies that the data is to be written into the addressed memory location. MWTC/ is asynchronous with respect to BCLK/.

IORC/

I/O read command; indicates that the address of an input port has been placed on the system address bus and that the data (8 or 16 bits) at that input port is to be read and placed on the system data bus. IORC/ is asynchronous with respect to BCLK/.

IOWC/

I/O write command; indicates that the address of an output port has been placed on the system address bus and that the contents of the system data bus (8 or 16 bits) are to be output to the address port. IOWC/ is asynchronous with respect to BCLK/.

XACK/

Transfer acknowledge signal; the required response of a slave board which indicates that the specified read/write operation has been completed. That is, data has been placed on, or accepted from, the system data bus lines. XACK/ is asynchronous with respect to BCLK/.

Asynchronous Interrupt Lines

INT0/ - INT7/

8 Multi-level, parallel interrupt request lines;

used with a parallel interrupt resolution network. INT0/ has the highest priority, while INT7/ has lowest priority. Interrupt lines should be driven with open collector drivers.

INTA/

Interrupt acknowledge; an interrupt acknowledge line (INTA/), driven by the bus master, requests the transfer of interrupt information onto the bus from slave priority interrupt controllers (8259s or 8259As). The specific information timed onto the bus depends upon the implementation of the interrupt scheme. In general, the leading edge of INTA/ indicates that the address bus is active while the trailing edge indicates that data is present on the data lines.

MULTIBUS P2 Signal Lines — The signals contained on the MULTIBUS P2 auxiliary connector are used primarily by optional power back-up circuitry for memory protection. P2 signals are not bused on the backplane, and therefore, require a separate connector for each board using the P2 signals. Present iSBC boards have a slot in the card edge and should be used with a keyed P2 edge connector. Use of the P2 signal lines is optional.

ACLO

AC Low; this signal generated by the power supply goes high when the AC line voltage drops below a certain voltage (e.g., 103v AC in 115v AC line voltage systems) indicating D.C. power will fail in 3 msec. ACLO goes low when all D.C. voltages return to approximately 95% of the regulated value. This line must be pulled up by the optional standby power source, if one is used.

PFIN/

Power fail interrupt; this signal interrupts the processor when a power failure occurs, it is driven by external power fail circuitry.

PFSN/

Power fail sense; this line is the output of a latch which indicates that a power failure has occurred. It is reset by PFSR/. The power fail

sense latch is part of external power fail circuitry and must be powered by the standby power source.

PFSR/

Power fail sense reset; this line is used to reset the power fail sense latch (PFSN/).

MPRO/

Memory protect; prevents memory operation during period of uncertain DC power, by inhibiting memory requests. MPRO/ is driven by external power fail circuitry.

ALE

Address latch enable; generated by the CPU (8085 or 8086) to provide an auxiliary address latch.

HALT/

Halt; indicates that the master CPU is halted.

AUX RESET/

Auxiliary Reset; this externally generated signal initiates a power-up sequence.

WAIT/

Bus master wait state; this signal indicates that the processor is in a wait state.

Reserved — Several P1 and P2 connector bus pins are unused. However, they should be regarded as reserved for dedicated use in future Intel products.

Power Supplies — The power supply bus pins are detailed in Appendix A which contains the pin assignment of signals on the MULTIBUS backplane.

It is the designer's responsibility to provide adequate bulk decoupling on the board to avoid current surges on the power supply lines. It is also recommended that you provide high frequency

decoupling for the logic on your board. Values of 22uF for +5v and +12v pins and 10uF for –5v and –12v pins are typical on iSBC boards.

Operating Characteristics

Beyond the definition of the MULTIBUS signals themselves, it is important to examine the operating characteristics of the bus. The AC requirements outline the timing of the bus signals and in particular, define the relationships between the various bus signals. On the other hand, the DC requirements specify the bus driver character- istics, maximum bus loading per board, and the pull-up/down resistors.

The AC requirements are best presented by a discussion of the relevant timing diagrams. Appendix B contains a list of the MULTIBUS timing specifications. The following sections will discuss data transfers, inhibit operations, inter- rupt operations, MULTIBUS multi-master opera- tion and power fail considerations.

Data Transfers — Data transfers on the MULTI- BUS system bus occur with a maximum band- width of 5 MHz for single or multiple read/write transfers. Due to bus arbitration and memory access time, a typical maximum transfer rate is often on the order of 2 MHz.

Read Data

Figure 1 shows the read operation AC timing diagram. The address must be stable (t_{AS}) for a minimum of 50 ns before command (IORC/ or MRDC/). This time is typically used by the bus interface to decode the address and thus provide the required device selects. The device selects establish the data paths on the user system in anticipation of the strobe signal (command) which will follow. The minimum command pulse width is 100 ns. The address must remain stable for at least 50 ns following the command (t_{AH}). Valid data should not be driven onto the bus prior to command, and must not be removed until the command is cleared. The XACK/ signal, which is a response indicating the specified read/write operation has been completed, must coincide or follow both the read access and valid data (t_{DXL}). XACK/ must be held until the command is cleared (t_{XAH}).
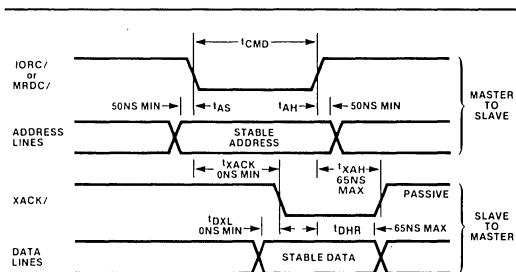
Figure 1. Read AC Timing

Write Data

The write operation AC timing diagram is shown in Figure 2. During a write data transfer, valid data must be presented simultaneously with a stable address. Thus, the write data setup time (t_{DS}) has the same requirement as the address setup time (t_{AS}). The requirement for stable data both before and after command (IOWC/ or MWTC/) enables the bus interface circuitry to latch data on either the leading or trailing edge of command.
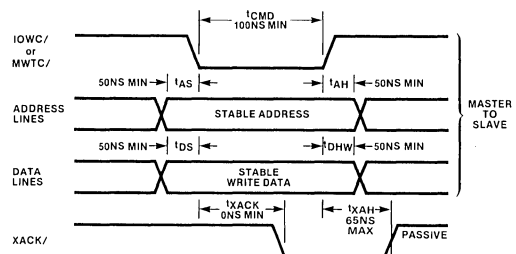
Figure 2. Write AC Timing

Data Byte Swapping in 16-bit Systems

A 16-bit master may transfer data on the MULTI- BUS data lines using 8-bit or 16-bit paths depending on whether a byte or word (2 byte) operation has been specified. (A word transfer specified with an odd I/O or memory address will actually be executed as two single byte transfers.) An 8-bit master may only perform byte transfers on the MULTIBUS data lines DAT0/ - DAT7/.

In order to maintain compatibility with older 8-bit masters and slaves, a byte swapping buffer is included in all new 16-bit masters and 16-bit slaves. In the iSBC product line, all byte transfers will take place on the low 8 data lines DAT0/ - DAT7/. Figure 3 contains a example of 8/16-bit

data driver logic for 16-bit master and slave systems. In the 8/16-bit system, there are three sets of buffers; the lower byte buffer which accesses DAT0/ - DAT7/, the upper byte buffer which accesses DAT8/ - DATF/, and the swap byte buffer which accesses the MULTIBUS data lines DAT0/ - DAT7/ and transfers the data to/from the on-board data bus lines D8 - DF.

Figure 4 summarizes the 8 and 16-bit data paths used for three types of MULTIBUS transfers. Two signals control the data transfers.

Byte High Enable (BHEN/) active indicates that the bus is operating in sixteen bit mode, and Address Bit 0 (ADR0/) defines an even or odd byte transfer address.

On the first type of transfer, BHEN/ is inactive, and ADR0/ is inactive indicating the transfer of an even eight bit byte. The transfer takes place across data lines DAT0/ - DAT7/.

On the second type of transfer, BHEN/ is inactive, and ADR0/ is active indicating the transfer of a high (odd) byte. On this type of transfer, the odd (high) byte is transferred through the Swap Byte Buffer to DAT0/ - DAT7/. This makes eight bit and sixteen bit systems compatible.
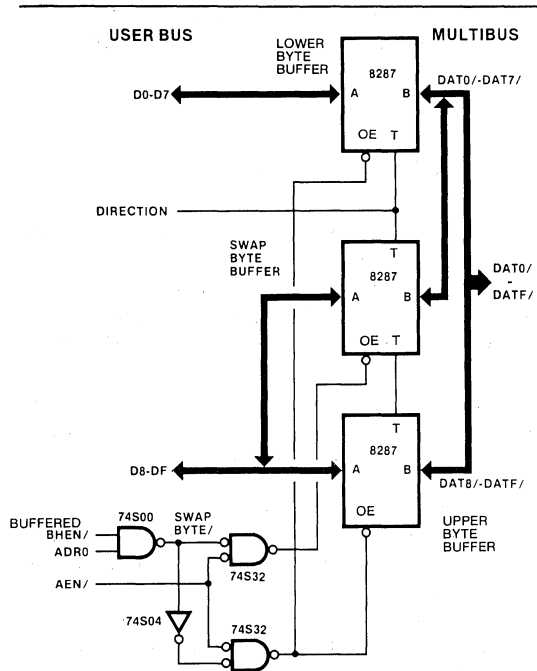
Figure 3. 8/16-Bit Data Drivers

16-BIT DEVICE	MULTIBUS	BHEN/	ADR0/	MULTIBUS TRANSFER DATA PATH	DEVICE BYTE TRANSFERRED
LOW, EVEN BYTES → DAT0/ - DAT7/ HIGH,ODD BYTES → DAT8/ - DATF/		H	H	8-BIT, DAT0/ - DAT7/	EVEN
LOW, EVEN BYTES → DAT0/ - DAT7/ HIGH,ODD BYTES → DAT8/ - DATF/		H	L	8-BIT, DAT0/ - DAT7/	ODD
LOW, EVEN BYTES → DAT0/ - DAT7/ HIGH, ODD BYTES → DAT8/ - DATF/		L	H	16-BIT, DAT0/ - DATF/	EVEN AND ODD

Figure 4. 8/16-Bit Device Transfer Operation

The third type of transfer is a 16 bit (word) transfer. This is indicated by BHEN/ being active, and ADR0/ being inactive. On this type of transfer, the low (even) byte is transferred on DAT0/ - DAT7/ and the high (odd) byte is transferred on DAT8/ - DATF/.

Note that the condition when both BHEN/ and ADR0/ are active is not used with present iSBC boards. This condition could be used to transfer a high odd byte of data on DAT8/ - DATF/, thus eliminating the need for the swap byte buffer. However, this is not a recommended transfer type, because it eliminates the capability of communicating with 8-bit modules.

Inhibit Operations — Bus inhibit operations are required by certain bootstrap and memory mapped I/O configurations. The purpose of the inhibit operation is to allow a combination of RAM, ROM, or memory mapped I/O to occupy the same memory address space. In the case of a bootstrap, it may be desirable to have both ROM and RAM memory occupy the same address space, selecting ROM instead of RAM for low order memory only when the system is reset. A system designed to use

memory mapped I/O, which has actual memory occupying the memory mapped I/O address space, may need to inhibit RAM or ROM memory to perform its functions.

There are two essential requirements for a successful inhibit operation. The first is that the inhibit signal must be asserted as soon as possible, within a maximum of 100 ns (t_{CI}), after stable address. The second requirement for a successful inhibit operation is that the acknowledge must be delayed (t_{XACKB}) to allow the inhibited slave to terminate any irreversible timing operations initiated by detection of a valid command prior to its inhibit.

This situation may arise because a command can be asserted within 50 ns after stable address (t_{AS}) and yet inhibit is not required until 100 ns (t_{ID}) after stable address. The acknowledge delay time (t_{XACKB}) is a function of the cycle time of the inhibited slave memory. Inhibiting the iSBC 016 RAM board, for example, requires a minimum of 1.5 usec. Less time is typically needed to inhibit other memory modules. For example, the iSBC 104 board requires 475 ns.

Figure 5 depicts a situation in which both RAM
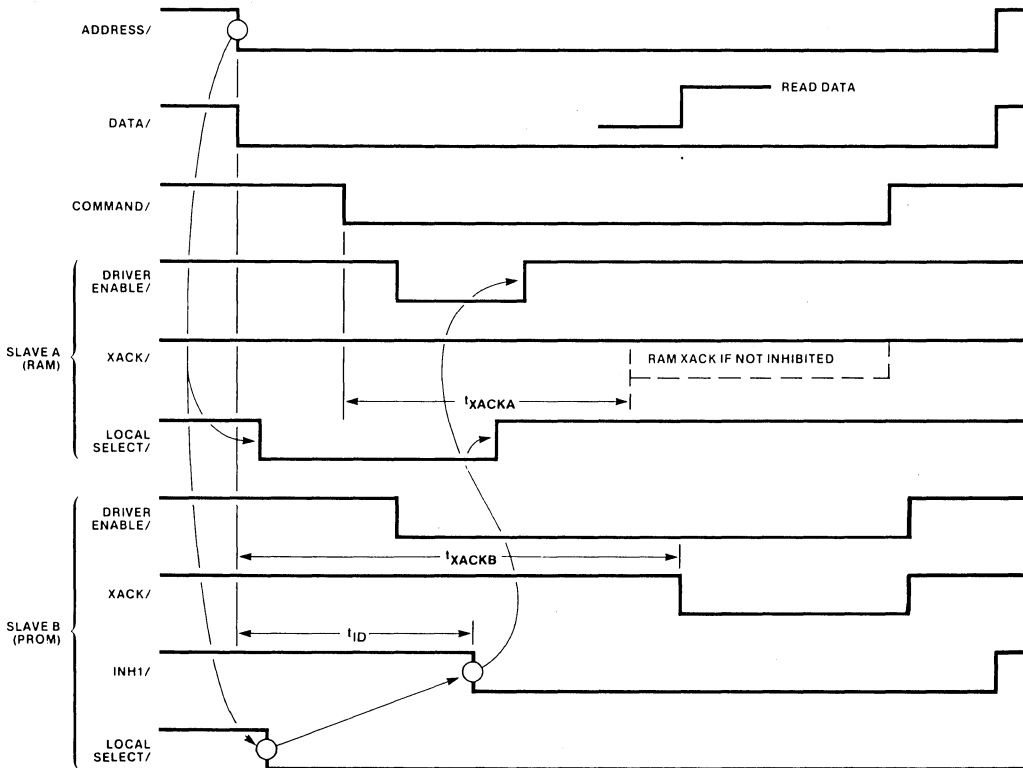
Figure 5. Inhibit Timing

and PROM memory have the same memory addresses. In this case, PROM inhibits RAM, producing the effect of PROM overriding RAM. After address is stable, local selects are generated for both the PROM and the RAM. The PROM local select produces the INH1/ signal which then removes the RAM local select and its driver enable. Because the slave RAM has been inhibited after it had already begun its cycle, the PROM XACK/ must be delayed (t_{XACKB}) until after the latest possible acknowledgement from the RAM (t_{XACKA}).

Interrupt Operations — The MULTIBUS interrupt lines INT0/ - INT7/ are used by a MULTIBUS master to receive interrupts from bus slaves, other bus masters or external logic such as power fail logic. A bus master may also contain internal interrupt sources which do not require the bus interrupt lines to interrupt the master. There are two interrupt implementation schemes used by bus interrupts, Non Bus Vectored Interrupts and Bus Vectored Interrupts. Non Bus Vectored Interrupts do not convey interrupt vector address information on the bus. Bus Vectored Interrupts are interrupts from slave Priority Interrupt Controllers (PICs) which do convey interrupt vector

address information on the bus.

Non Bus Vectored Interrupts

Non Bus Vectored Interrupts are those interrupts whose interrupt vector address is generated by the bus master and do not require the MULTIBUS address lines for transfer of the interrupt vector address. The interrupt vector address is generated by the interrupt controller on the master and transferred to the processor over the local bus. The source of the interrupt can be on the master module or on other bus modules, in which case the bus modules use the MULTIBUS interrupt request lines (INT0/ - INT7/) to generate their interrupt requests to the bus master. When an interrupt request line is activated, the bus master performs it own interrupt operation and processes the interrupt. Figure 6 shows an example of Non Bus Vectored Interrupt implementation.

Bus Vectored Interrupts

Bus Vectored Interrupts (Figure 7) are those interrupts which transfer the interrupt vector address along the MULTIBUS address lines from the slave to the bus master using the INTA/ command signal for synchronization.
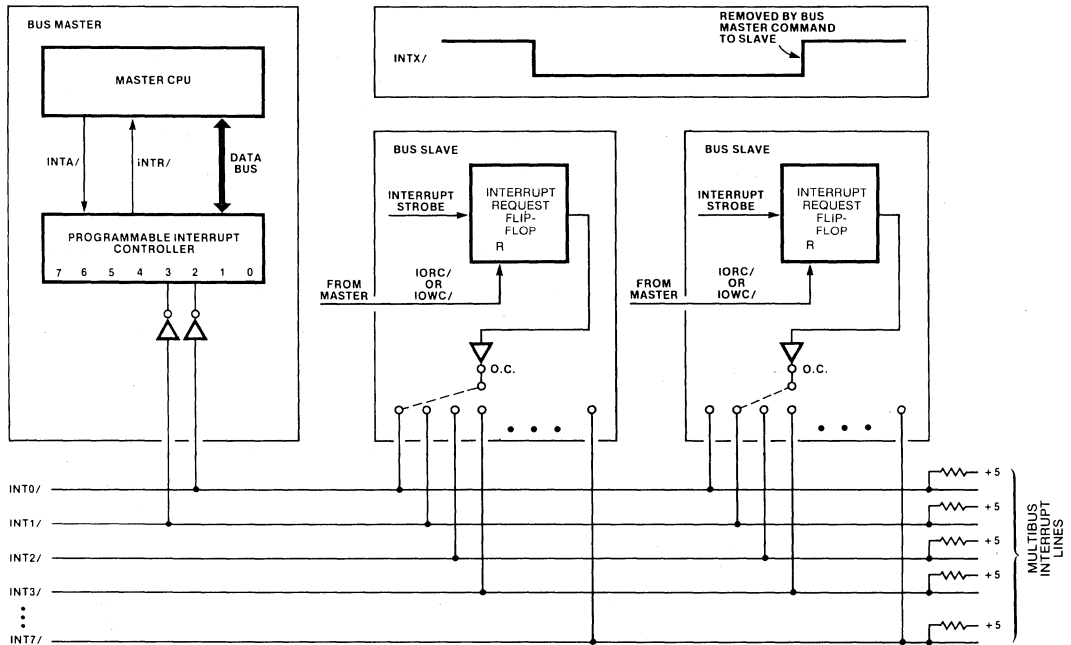
Figure 6. Non Bus Vectored Interrupt Implementation
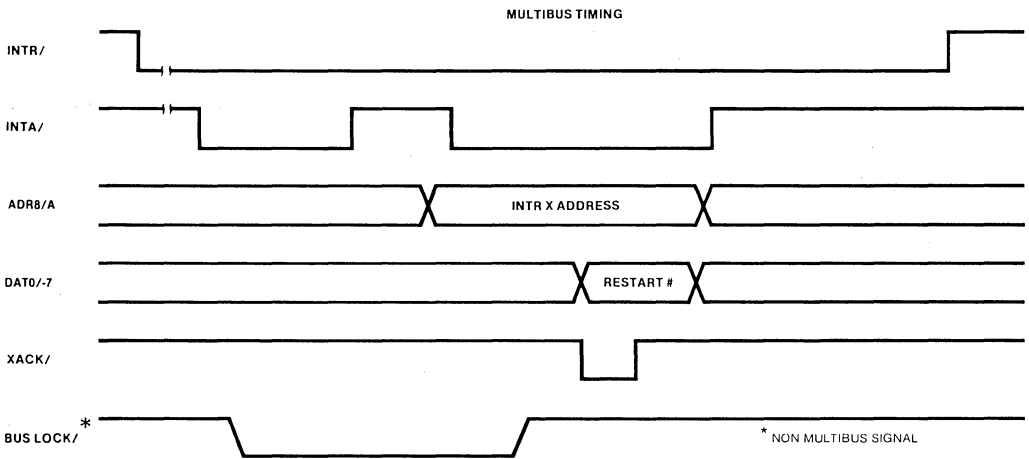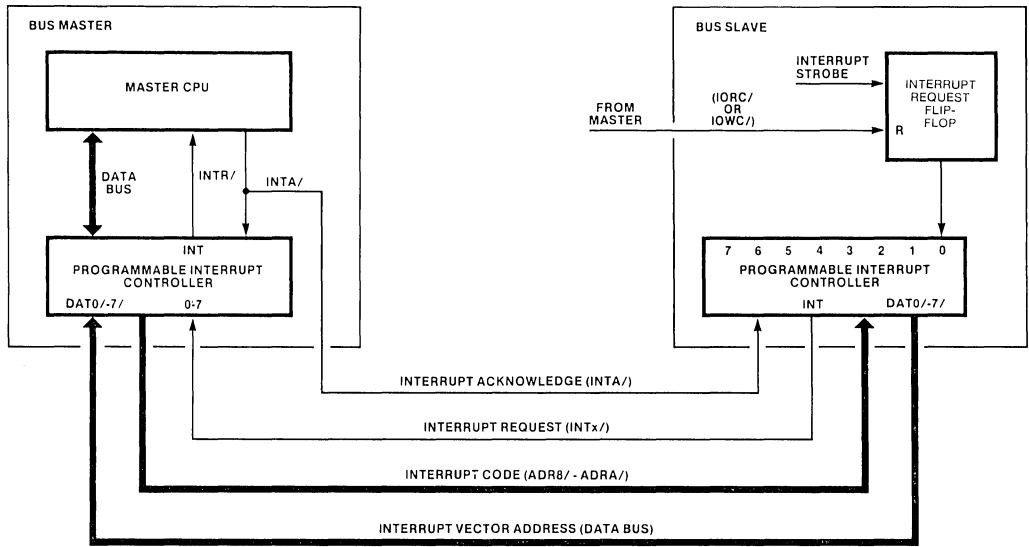
Figure 7. Bus Vectored Interrupt Logic (With 2 INTA/ Timing Diagram)

When an interrupt request from the MULTIBUS interrupt lines INT0/ - INT7/ occurs, the interrupt control logic on the bus master interrupts its processor. The processor on the bus master generates an INTA/ command which freezes the state of the interrupt logic on the MULTIBUS slaves for priority resolution. The bus master also locks (retains the bus between bus cycles) the MULTIBUS control lines to guarantee itself consecutive bus cycles. After the first INTA/ command, the bus master's interrupt control logic puts an interrupt code on to the MULTIBUS address lines ADR8/ - ADRA/. The interrupt code is the address of the highest priority active interrupt request line. At this point in the Bus Vectored

Interrupt procedure, two different sequences could take place. The difference occurs, because the MULTIBUS specification can support masters which generate one additional INTA/ (8086 masters) or two additional INTA/s (8080A and 8085 masters).

If the bus master generates one additional INTA/, this second INTA/ causes the bus slave interrupt control logic to transmit an interrupt vector 8-bit pointer on the MULTIBUS data lines. The vector pointer is used by the bus master to determine the memory address of the interrupt service routine.

If the bus master generates two additional INTA/s, these two INTA/ commands allow the

bus slave to put a two byte interrupt vector address on to the MULTIBUS data lines (one byte for each INTA/). The interrupt vector address is used by the bus master to service the interrupt.

The MULTIBUS specification provides for only one type of Bus Vectored Interrupt operation in a given system. Slave boards which have an 8259 interrupt controller are only capable of 3 INTA/ operation (2 additional INTA/s after the first INTA/). Slave boards with the 8259A interrupt controller are capable of either 2 INTA/ or 3 INTA/ operation. All slave boards in a given system must operate in the same way (2 INTA/s or 3 INTA/s) if Bus Vectored Interrupts are to be used. However, the MULTIBUS specification does provide for Bus Vectored Interrupts and Non Bus Vectored Interrupts in the same system.

MULTIBUS Multi-Master Operation — The MULTIBUS system bus can accommodate several bus masters on the same system, each one taking control of the bus as it needs to affect data transfers. The bus masters request bus control through a bus exchange sequence.

Two bus exchange priority resolution techniques are discussed, a serial technique and a parallel technique. Figures 8 and 9 illustrate these two techniques. The bus exchange operation discussed later is the same for both techniques.

Serial Priority Technique

Serial priority resolution is accomplished with a daisy chain technique (see Figure 8). The priority input (BPRN/) of the highest priority master is tied to ground. The priority output (BPRO/) of the

highest priority master is then connected to the priority input (BPRN/) of the next lower priority master, and so on. Any master generating a bus request will set its BPRO/ signal high to the next lower priority master. Any master seeing a high signal on its BPRN/ line will sets its BPRO/ line high, thus passing down priority information to lower priority masters. In this implementation, the bus request line (BREQ/) is not used outside of the individual masters. A limited number of masters can be accommodated by this technique, due to gate delays through the daisy chain. Using the current Intel MULTIBUS controller chip on the master boards up to 3 masters may be accommodated if a BCLK/ period of 100 ns is used. If more bus masters are required, either BCLK/ must be slowed or a parallel priority technique used.

Parallel Priority Technique

In the parallel priority technique, the priority is resolved in a priority resolution circuit in which the highest priority BREQ/ input is encoded with a priority encoder chip (74148). This coded value is then decoded with a priority decoder chip (74S138) to activate the appropriate BPRN/ line. The BPRO/ lines are not used in the parallel priority scheme. However, since the MULTIBUS backplane contains a trace from the BPRN/ signal of one card slot to the BPRO/ signal of the adjacent lower card slot, the BPRO/ must be disconnected from the bus on the board or the backplane trace must be cut. A practical limit of sixteen masters can be accommodated using the parallel priority technique due to physical bus length limitations. Figure 9 contains the schematic for a typical parallel resolution network. Note that the parallel priority resolution network must be externally supplied.
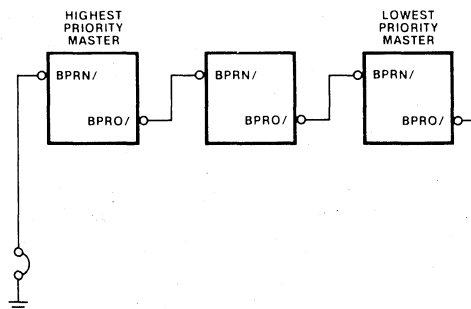
Figure 8. Serial Priority Technique
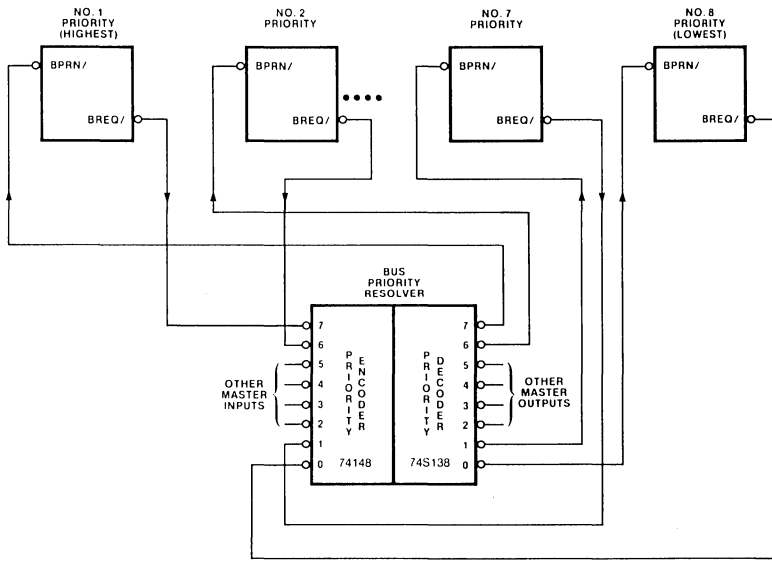
Figure 9. Parallel Priority Technique

MULTIBUS Exchange Operation — A timing diagram for the MULTIBUS exchange operation is shown in Figure 10. This implementation example uses a parallel resolution scheme, however, the timing would be basically the same for the serial resolution scheme.

In this example, master A has been assigned a lower priority than master B. The bus exchange occurs because master B generates a bus request during a time when master A has control of the bus.

The exchange process begins when master B requires the bus to access some resource such as an I/O or memory module while master A controls the bus. This internal request is synchronized with the trailing edge (high to low) of BCLK/ to generate a bus request (BREQ/). The bus priority resolution circuit changes the BPRN/ signal from active (low) to inactive (high) for master A and from inactive to active for master B. Master A must first complete the current bus command if one is in operation. After master A completes the command, it sets BUSY/ inactive on the next trailing edge of BCLK/. This allows the actual bus exchange to occur, because master A has relinquished control of the bus, and master B has been granted its BPRN/. During this time, the drivers

for master A are disabled. Master B must take control of the bus with the next trailing edge of BCLK/ to complete the bus exchange. Master B takes control by activating BUSY/ and enabling its drivers.

It is possible for master A to retain control of the bus and prevent master B from getting control. Master A activates the Bus Override (or Bus Lock) signal which keeps BUSY/ active allowing control of the bus to stay with master A. This guarantees a master consecutive bus cycles for software or hardware functions which require exclusive, continuous access to the bus.

Note that in systems with only a single master it is necessary to ground the BPRN/ pin of the master, if slave boards are to be accessed. In single board systems which use a CPU board capable of Bus Vectored Interrupt operation, the BPRN/ pin must also be grounded.

In a single master system bus transfer efficiency may be gained if the BUS OVERRIDE signal is kept active continuously. This permits the master to maintain control of the bus at all times, therefore saving the overhead of the master reacquiring the bus each time it is needed.

The CBRQ/ line may be used by a master in control of the bus to determine if another master
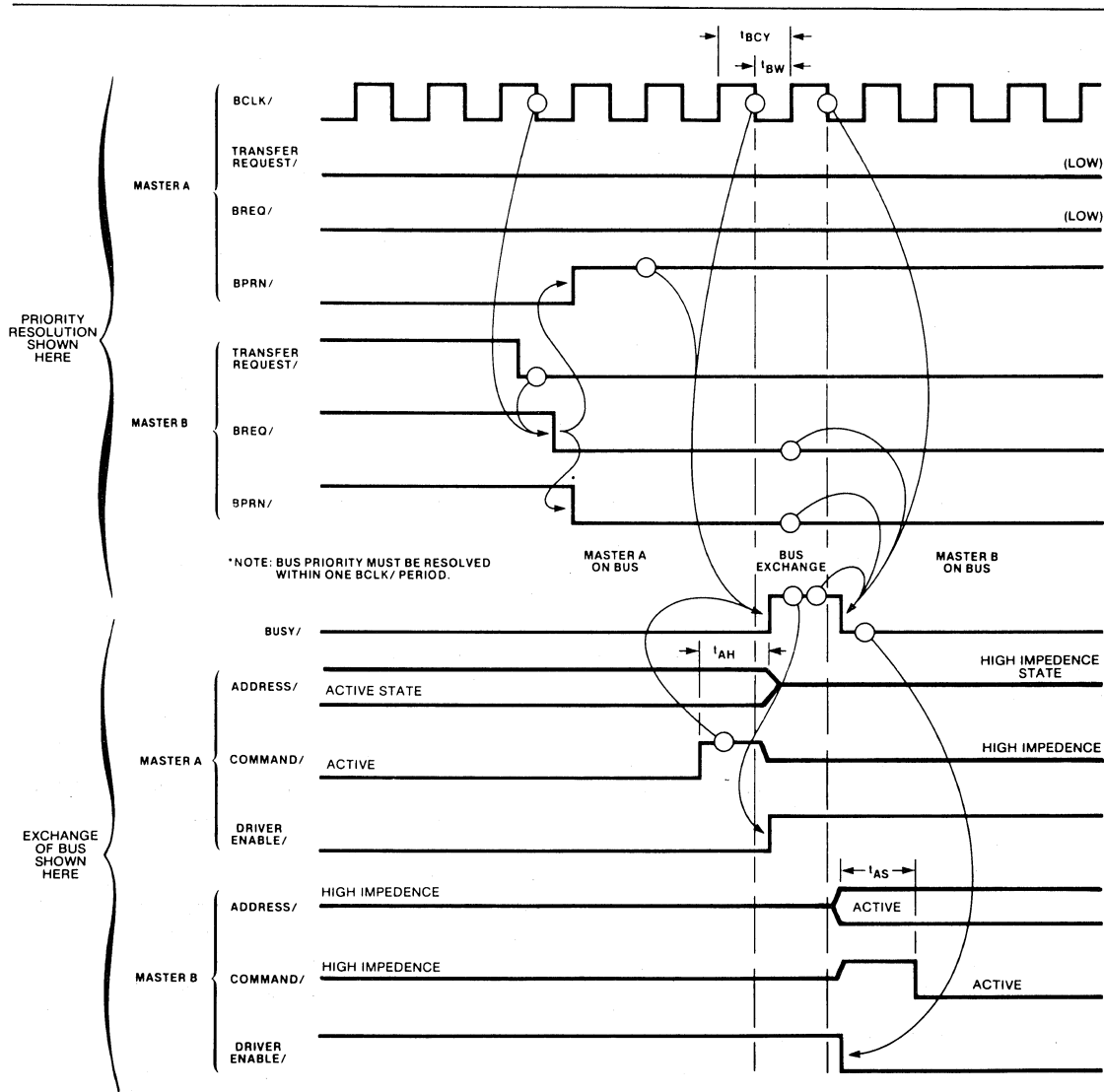
Figure 10. Bus Control Exchange Operation

requires the bus. If a master currently in control of the bus sees the CBRQ/ line inactive, it will maintain control of the bus between adjacent bus accesses. Therefore, when a bus access is required, the master saves the overhead of reacquiring the bus. If a current bus master sees the CBRQ/ line active, it will then relinquish control of the bus after the current bus access and will contend for the bus with the other master(s) requiring the bus. The relative priorities of the masters will determine which master receives the bus.

Note that except for the BUS OVERRIDE state, no single master may keep exclusive control of the bus. This is true because it is impossible for the CPU on a master to require continuous access to the bus. Other lower priority masters will always be able to gain access to the bus between accesses of a higher priority master.

Power Fail Considerations — The MULTIBUS P2 connector signals provide a means of handling power failures. The circuits required for power
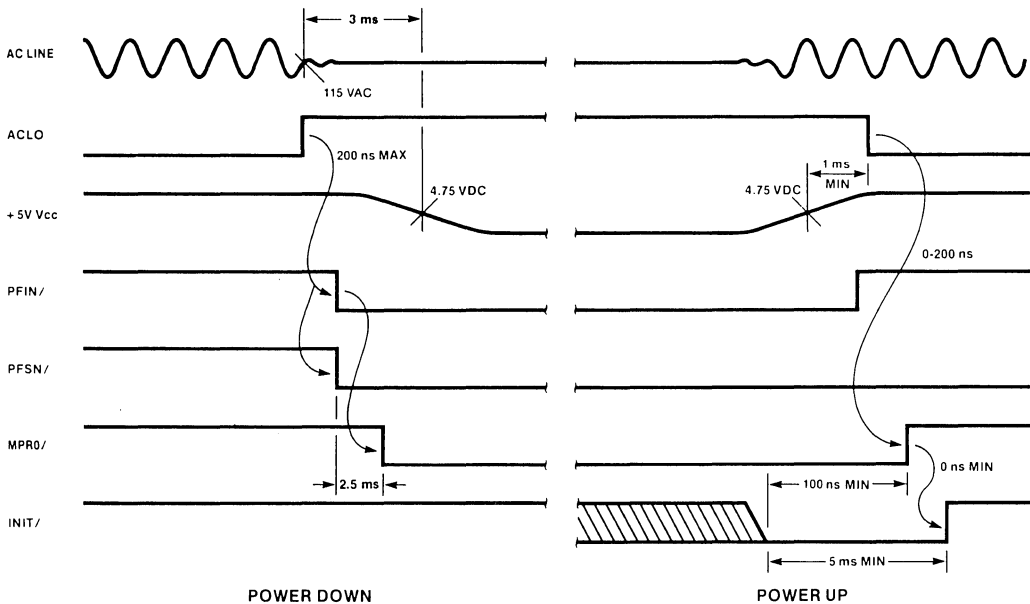
Figure 11. Power Fail Timing Sequence

failure detection and handling are optional and must be supplied by the user. Figure 11 shows the timing of a power fail sequence.

The power supply monitors the AC power level. When power drops below an acceptable value, the power supply raises ACLO which tells the power fail logic that a minimum of three milliseconds will elapse before DC power will fall below regulated voltage levels. The power fail logic sets a sense latch (PFSN/) and generates an interrupt (PFIN/) to the processor so the processor can store its environment. After a 2.5 millisecond timeout, the memory protect signal (MPRO/) is asserted by the power fail logic preventing any memory activity. As power falls, the memory goes on standby power. Note that the power fail logic must be powered from the standby source.

As the AC line revives, the logic voltage level is monitored by the power supply. After power has been at its operating level for one millisecond minimum, the power supply sets the signal ACLO low, beginning the restart sequence. First, the memory protect line (MPRO/) then the initialize line (INIT/) become inactive. The bus master now starts running. The bus master checks the power fail latch (PFSN/) and, if it finds it set, branches to

a power up routine which resets the latch (PFSR/), restores the environment, and resumes execution.

Note that INIT/ is activated only after DC power has risen to the regulated voltage levels and must stay low for five milliseconds minimum before the system is allowed to restart. Alternatively, INIT/ may be held low through an open collector device by MPRO/.

How the power failure equipment is configured is left to the system designer. The backup power source may be batteries located on the memory boards or more elaborate facilities located off-board. The location of the power fail logic determines which MULTIBUS power fail lines are used. Pins on the P2 connector have been specified for the power failure functions for use as needed.

To further clarify the location and use of the power fail circuitry, an example of a typical power fail system block diagram is shown in Figure 12. A single board computer and a slave memory board are contained in the system. It is desired to power the memory circuit elements of the memory board from auxiliary power. The single board computer will remain on the main power supply. To accomplish this, user supplied power fail logic and
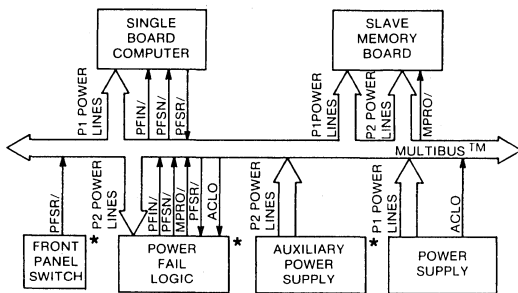
* USER SUPPLIED

Figure 12. Typical Power Fail System Block Diagram

an auxiliary power supply have been included in the system.

The single board computer is powered from the P1 power lines and accesses the P2 signal lines PFIN/, PFSN/ and PFSR/ (only the P2 signal lines used by a particular functional block are shown on the block diagram). The PFSR/ line is driven from two sources: a front panel switch and the single board computer. The front panel switch is used during normal power-up to reset the power fail sense latch. The single board computer uses the PFSR/ line to reset the latch during a power-up sequence after a power failure. Current single board computers must access the PFSN/ and PFSR/ signals either directly with dedicated circuitry and a P2 pin connection or through the parallel I/O lines with a cable connection from the parallel I/O connector to the P2 connector.

The slave memory board uses both the P1 and P2 power lines, the P2 power lines are used (at all times) to power the memory circuit elements and other support circuits, the P1 power lines power all other circuitry. In addition, the MPRO/ line is input and used to sense when memory contents should be protected.

The power fail logic contains the power fail sense latch, and uses the PFSR/ and ACLO lines for inputs and the PFIN/ PFSN/, and MPRO/ lines for outputs. The power fail logic must be powered by the P2 power lines.

DC Requirements — The drive and load characteristics of the bus signals are listed in Appendix C. The physical locations of the drivers and loads, as well as the terminating resistor value for each bus line, are also specified. Appendix D contains the MULTIBUS power specifications.

MULTIBUS™ Slave Interface Circuit Elements

There are three basic elements of a slave bus interface: address decoders, bus drivers, and control signal logic. This section discusses each of these elements in general terms. A description of a detailed implementation of a slave interface is presented in a later section of this application note.

Address Decoding — This logic decodes the appropriate MULTIBUS address bits into RAM requests, ROM requests, or I/O selects. Care must be taken in the design of the address decode logic to ensure flexibility in the selection of base address assignments. Without this flexibility, restrictions may be placed upon various system configurations. Ideally, switches and jumper connections should be associated with the decode logic to permit field modification of base address assignments.

The initial step in designing the address decode portion of a MULTIBUS interface is to determine the required number of unique address locations. This decision is influenced by the fact that address decoding is usually done in two stages. The first stage decodes the base address, producing an enable for the second stage which generates the actual device selects for the user logic. A convenient implementation of this two stage decoding scheme utilizes a pair of decoders driven by the high order bits of the address for the first stage and a second decoder for the low order bits of the address bus. This technique forces the number of unique address locations to be a power of two, based at the address decoded by the first stage. Consider the scheme illustrated in Figure 13.

As shown in Figure 13, the address bits A_4 - A_B are used to produce switch selected outputs of the first stage of decoding. The 1 out of 8 binary decoders

have been used. The top decoder decodes address lines A_4 - A_7, and the bottom decoder decodes address lines A_8 - A_B. If only address lines A_0 - A_7 are being used for device selection, as in the case of I/O port selection in 8-bit systems, the bottom decoder may be disabled by setting switch S2 to the ground position. Address lines A_7 and A_B drive enable inputs E2 or E3 of the decoders. The address lines A_0 - A_3 enter the second stage address decoder to produce 8 user device selects. The second stage decoder must first be enabled by an address that corresponds to the switch-selected base address.

Address decoding must be completed before the arrival of a command. Since the command may become active within 50 ns after stable address, the decode logic should be kept simple with a minimal number of layers of logic. Furthermore, the timing is extremely critical in systems which make use of the inhibit lines.

A linear or unary select scheme in which no binary encoding of device address (e.g., address bit A_0 selects device 0, address bit A_1 selects device 1, etc.) is performed is not recommended because the scheme offers no protection in case multiple devices are simultaneously selected, and because the addressing within such a system is restricted by the extent of the address space occupied by such a scheme.
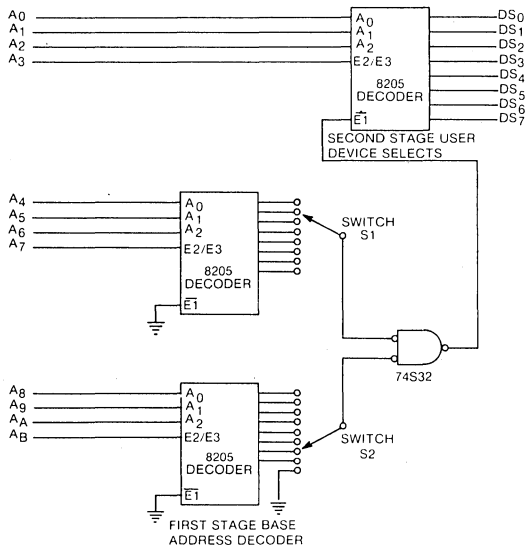
Data Bus Drivers — For user designed logic which simply receives data from the MULTIBUS data lines, this portion of the bus interface logic may only consist of buffers. Buffers are required to ensure that maximum allowable bus loading is not exceeded by the user logic.

In systems where the user designed logic must place data onto the MULTIBUS data lines, three-state drivers are required. These drivers should be enabled only when a memory read command (MRDC/) or an I/O read command (IORC/) is present and the module has been addressed.

When both the read and write functions are required, parallel bidirectional bus drivers (e.g., Intel 8226, 8287, etc.) are used. A note of caution must be included for the designer who uses this type of device. A problem may arise if data hold time requirements must be satisfied for user logic following write operations. When bus commands are used to directly produce both the chip select for the bidirectional bus driver and a strobe to a latch in the user logic, removal of that signal may not provide the user's latch with adequate data hold time. Depending on the specifics of the user logic, this problem may be solved by permanently enabling the data buffer's receiver circuits and controlling only the direction of the buffers.

Control Signal Logic — The control signal logic consists of the circuits that forward the I/O and memory read/write commands to their respective destinations, provide the bus with a transfer acknowledge response, and drive the system interrupt lines.

Bus Command Lines

The MULTIBUS information transfer protocol lines (MRDC/, MWTC/, IORD/. and IOWC/) should be buffered by devices with very high speed switching. Because the bus DC requirements specify that each board may load these lines with 2.0 mA, Schottky devices are recommended. LS devices are not recommended due to their poor noise immunity. The commands should be gated

Figure 13. Two Stage Decoding Scheme

with a signal indicating the base address has been decoded to generate read and write strobes for the user logic.

Transfer Acknowledge Generation

The user interface transfer acknowledge generation logic provides a transfer acknowledge response, XACK/, to notify the bus master that write data provided by the bus master has been accepted or that read data it has requested is available on the MULTIBUS data lines. XACK/ allows the bus master to conclude its current instruction.

Since XACK/ timing requirements depend on both the CPU of the bus master and characteristics of the user logic, a circuit is needed which will provide a range of easily modified acknowledge responses.

The transfer acknowledge signals must be driven by three-state drivers which are enabled when the bus interface is addressed and a command is present.

Interrupt Signal Lines

The asynchronous interrupt lines must be driven by open collector devices with a minimum drive of 16 mA.

In a typical Non Bus Vectored Interrupt system, logic must be provided to assert and latch-up an interrupt signal. In addition to driving the MULTIBUS interrupt lines, the latched interrupt signal would be read by an I/O operation such as reading the module's status. The interrupt signal would be cleared by writing to the status register.

III. MULTIBUS™ SLAVE DESIGN EXAMPLE

A MULTIBUS slave design example has been included in this application note to reinforce the theory previously discussed. The design example is of general purpose I/O slave interface. This design example could easily be modified to be used as a slave memory interface by buffering the address signals and using the appropriate MULTIBUS memory commands. In addition, to help the reader better understand an application for an I/O slave interface, two Intel 8255A Parallel Peripheral Interface (PPI) devices are shown connected to the slave interface.

The design example is shown in both 8/16-bit version and an 8-bit version. The 8/16-bit version

is an I/O interface which will permit a 16-bit master to perform 8 or 16 bit data transfers. 8-bit masters may also use the 8/16-bit version of the design example to perform 8-bit data transfers.

The 8-bit version of the design example may be used by both 8 or 16-bit masters, but will only perform 8-bit data transfers. It does not contain the circuitry required to perform 16-bit data transfers.

Both the 8/16-bit version and the 8-bit version of the design example were implemented on an iSBC 905 prototype board. The schematics for each of the examples are given in Appendices F and G.

Functional/Programming Characteristics

This section describes the organization of the slave interface from two points of view, the functional point of view and the programming characteristics. First, the principal functions performed by the hardware are identified and the general data flow is illustrated. This point of view is intended as an introduction to the detailed description provided in the next section; Theory of Operation. In the second point of view, the information needed by a programmer to access the slave is summarized.

Functional Description — The function of this I/O slave is to provide the bus interface logic for general purpose I/O functions and for two Intel 8255A Parallel Peripheral Interface (PPI) devices. Eight device selects (port addresses) are available for general purpose I/O functions. One of these device select lines is used to read and reset the state of an interrupt status flip-flop, the other seven device selects are unused in this design. An additional eight I/O device port addresses are used by the two 8255A devices; four I/O port addresses per 8255A (three I/O port address for the three parallel ports A, B, and C and the fourth I/O port address for the device control register).

Figure 14 contains a functional block diagram of the slave design example. This block diagram shows the fundamental circuit elements of a bus slave: bidirectional data bus drivers/receivers, address decoding logic and bus control logic. Also shown is the address decoding logic for the low order four bits, the interrupt logic which is selected by this decoding logic, and the two 8255A devices.
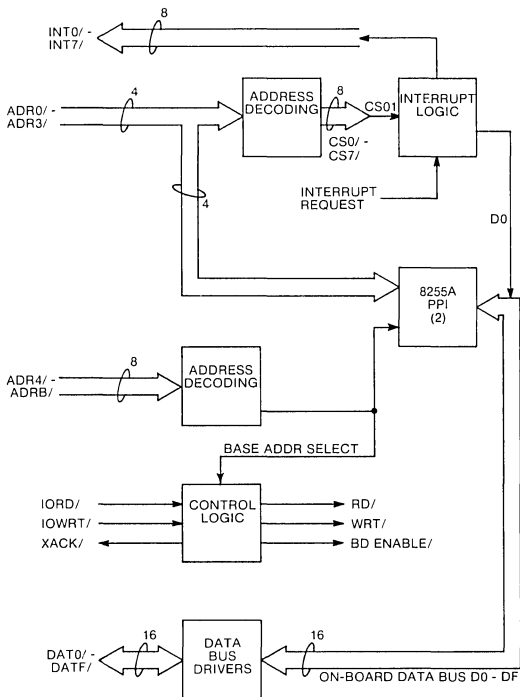
**Figure 14. MULTIBUS™ Slave Design Example
Functional Block Diagram**

Programming Characteristics — The slave design example provides 16 I/O port addresses which may be accessed by user software. The base address of the 16 contiguous port addresses is selected by wire wrap connections on the proto-type board. The wire wrap connections specify address bits ADR4/ - ADRB/. They allow the selection of a base address on any 16 byte boundary. Twelve address bits (ADR0/ - ADRB/) are used since 16-bit (8086 based) masters use 12 bits to specify I/O port addresses. If an 8 bit (8080 or 8085 based) master is used with this slave board, the high order address bits (ADR8/ - ADRB/) must not be used by the decoding circuits; a wire wrap jumper position (ground position) is provided for this.

The 16 I/O port addresses are divided into two groups of 8 port addresses by decoding address line ADR3/. Port addresses XX0 - XX7 are used for general I/O functions (XX indicates any hexi-decimal digit combination). Port address XX0 is used for accessing the interrupt status flip-flop and

port addresses XX1 - XX7 are not used in this example. Port addresses XX8 - XXF are used for accessing the PPIs. If port addresses XX8 - XXF are selected, then ADR0/ is used to specify which of two PPIs are selected. If the address is even (XX8, XXA, XXC, or XXE) then one PPI is selected. If the address is odd (XX9, XXB, XXD, or XXF), then the other PPI is selected. ADR1/ and ADR2/ are connected directly to the PPIs. Table 1 summarizes the I/O port addresses of the slave design example. Note that if a 16-bit master is used, it is possible to access the slave in a byte or word mode. If word access is used with port address XX8, XXA, XXC, or XXE, then 16 bit transfers will occur between the PPIs and the master. These 16 bit transfers occur because an even address has been specified and the MULTI-BUS BHEN/ signal indicates that a 16-bit transfer is requested.

Theory of Operation

In the preceding section, each of the slave design example functional blocks was identified and briefly explained. This section explains how these functions are implemented. For detailed circuit information, refer to the schematics in Appendices F and G. The schematic in Appendix F is on a foldout page so that the following text may easily be related to the schematic.

The discussion of the theory of operation is divided into five segments, each of which discusses a different function performed by the MULTIBUS slave design example. The five segments are:

1. Bus address decoding
2. Data buffers
3. Control signals
4. Interrupt logic
5. PPI operation

Each of these topics are discussed with regard to the 8/16-bit version of the design example; followed by a discussion of the circuit elements which are required by the 8-bit version of the interface.

Bus Address Decoding — Bus address decoding is performed by two 8205 1 out of 8 binary decoders. One decoder (A3) decodes address bits ADR8/ - ADRB/ and the second decoder (A2) decodes address bits ADR4/ - ADR7/. The base address

Table 1

SLAVE DESIGN EXAMPLE PORT ADDRESSES

I/O PORT ADDRESS	READ	WRITE
BYTE ACCESS		
XX0	Bit 0 = Interrupt Status	Reset Interrupt Status
XX1 - XX7	Unused	Unused
XX8	Parallel Port A, Even PPI	Parallel Port A, Even PPI
XX9	Parallel Port A, Odd PPI	Parallel Port A, Odd PPI
XXA	Parallel Port B, Even PPI	Parallel Port B, Even PPI
XXB	Parallel Port B, Odd PPI	Parallel Port B, Odd PPI
XXC	Parallel Port C, Even PPI	Parallel Port C, Even PPI
XXD	Parallel Port C, Odd PPI	Parallel Port C, Odd PPI
XXE	Illegal Condition	Control, Even PPI
XXF	Illegal Condition	Control, Odd PPI
WORD ACCESS		
XX0	Bit 0 = Interrupt Status	Reset Interrupt Status
XX2 - XX6	Unused	Unused
XX8	Parallel Port A, Even and Odd PPIs	Parallel Port A, Even and Odd PPIs
XXA	Parallel Port B, Even and Odd PPIs	Parallel Port B, Even and Odd PPIs
XXC	Parallel Port C, Even and Odd PPIs	Parallel Port C, Even and Odd PPIs
XXE	Illegal Condition	Control, Even and Odd PPIs

XX = Any hex digits, assigned by jumpers; XX defines the base address.

selected is determined by the position of wire wrap jumpers. The outputs of the two decoders are ANDed together to form the BASE ADR SELECT/ signal. This signal specifies the base address for a group of 16 I/O ports. Using the wire wrap jumper positions shown in the schematic, a base address of E3 has been selected. Therefore, this MULTIBUS slave board will respond to I/O port addresses in the E30 - E3F range.

If this slave board is to be used with 8-bit MULTIBUS masters, the high order address bits must not be decoded. Therefore, the wire wrap jumper which selects the output of decoder A3 must be placed in the top (ground) position (pin 10 of gate A9 to ground).

The low order 4 address lines (ADR0/ - ADR3/) are buffered and inverted using 74LS04 inverters. These address lines are input to an 8205 for decoding a chip select for the interrupt logic; the address lines are also used directly by the PPIs. LS-Series logic is required for buffering to meet the MULTIBUS specification for I_{IL} (low level input

current). S-Series or standard series logic will not meet this specification.

Address decoder A4 is used to decode addresses E30 - E37. The CS0/ output of this decoder is used to select the interrupt logic, thus I/O port address E30 is used to read and reset the interrupt latch. The remaining outputs from decoder A4 (CS1/ - CS7/) are not used in this example. They would normally be used to select other functions in a slave board with more capability. Note that in the schematic shown in Appendix G for the 8-bit version of this slave design example, the high order (ADR8/ - ADRB/) address decoder is not included and the BHEN/ signal is not used.

Data Buffers — Intel 8287 8-bit parallel bi-directional bus drivers are used for the MULTIBUS data lines DAT0/ - DATF/. In the 8/16-bit version of the slave board, three 8287 drivers are used.

When an 8-bit data transfer is requested, either driver A5, which is connected to on-board data

lines D0 - D7, or driver A6, which is connected to on-board data lines D8 - DF, is used. If a byte transfer is requested from an even address, driver A5 will be selected. If a byte transfer from an odd address is requested, driver A6 will be selected. All byte transfers take place on MULTIBUS data lines DAT0/ - DAT7/. When a word (16-bit) transfer is requested from an even address, drivers A5 and A7 will be used. Note that if a user program requests a word transfer from an odd address, 16-bit masters in the iSBC product line will actually perform two byte transfer requests.

The logic which determines the chip selection (8287 input signal OE, output enable) signals for the bus drivers uses the low order address bit (ADR0/) and the buffered Byte High Enable signal (BHENBL/). Note that the MULTIBUS signal BHEN/ has been buffered with an 74LS04 inverter. This is done to meet the bus address line loading specification. The SWAP BYTE/ signal which is generated is qualified by the BD ENBL/ signal and used to select the bus drivers.

The steering pin for the 8287 drivers is labelled T (transmit) and is driven by the signal RD. When an input (read) request is active or when neither a read or write command is being serviced, the direction of data transfer of the 8287 will be set for B to A.

The 8287 drivers are set to point IN (direction B to A) when no MULTIBUS I/O transfer command is being serviced for two reasons. First, if the driver were pointed OUT (direction A to B) and a write command occured, it would be necessary to turn the buffers IN and set the OE (output enable) signal active before the data could be transferred to the on-board bus. A possibility of a "buffer-fight" could occur in some designs if the OE signal permitted an 8287 to drive the MULTIBUS data lines momentarily before the steering signal could switch the direction of the 8287. In this case, both the MULTIBUS master and the slave would be driving the data lines; this is not recommended. (In this particular design, the steering signal will always stabilize before the OE signal becomes active.)

The second reason the driver is pointing IN when no command is present is due to the "data valid after WRITE" requirements of the 8255As. The 8255A requires that data remain on its data lines for 30 ns after the WRITE command ($\overline{WR}$ at the 8255A) is removed. This requirement will be met if the direction of the 8287 drivers is not switched

when the MULTIBUS IOWC/ signal is removed (WRT/ could have been used to steer the 8287 instead of RD); and if the capacitance of the on-board data bus lines is sufficient to hold the data values on the bus after the 8287 OE signal and the 8255A PPI WRT/ signal go inactive. The on-board data bus may easily be designed such that the capacitance of the lines is sufficient to meet the 30 ns data hold time requirement. In addition, the current leakage of all devices connected to the on-board bus must be kept small to meet the 30 ns data hold time requirement.

The 8-bit version of this design example uses only one 8287 instead of the three required by the 8/16-bit version. The logic required to control the swap byte buffer is also not necessary. The chip select signal used for the 8287 is the BD ENBL/ signal.

Control Signals — The MULTIBUS control signals used by this slave design example are IORC/, IOWC/, and XACK/. IORC/ and IOWC/ are qualified by the BASE ADR SELECT/ signal to form the signals RD and WRT. RD and WRT are used to drive the interrupt logic, the PPI logic and the XACK/ (transfer acknowledge) logic.

For the XACK/ logic RD and WRT are ORed to form the BD ENBL/ signal which is inverted and used to drive the CLEAR pin of a shift register. When the slave board is not being accessed, the CLEAR pin of the shift register will be low (BD ENBL/ is high). This causes the shift register to remain cleared and all outputs of the shift register will be low. When the slave board is accessed, the CLEAR pin will be high, and the A and B inputs (which are high) will be clocked to the output pins by CCLK/. To select a delay for the XACK/ signal, a jumper must be installed from one of the shift register output pins to the 8089 tri-state driver. Each of the shift register output pins select an integer multiple of CCLK/ periods for the signal delay. Since the CCLK/ signal is asynchronous, the actual delay selected may only be specified with a tolerance of one CCLK/ period. In this example a delay of 3 - 4 CCLK/ periods was selected; with a CCLK/ period of 100 ns, the XACK/ delay would occur somewhere within the range of 300 - 400 ns from the time when the CLEAR signal goes high.

The control signal logic used in the 8-bit version of the slave design example is identical to the logic used in the 8/16-bit version.

Interrupt Logic — The interrupt logic uses a 74S74 flip-flop to latch an asynchronous interrupt request from some external logic. The Q output of the INTERRUPT REQUEST LATCH is output through an open collector gate to one of the MULTIBUS interrupt lines. The state of the INTERRUPT REQUEST LATCH is transferred to the INTERRUPT STATUS LATCH when a read command is performed on I/O port BASE ADDRESS+0 (E30 for the jumper configuration shown). The Q output of INTERRUPT STATUS LATCH is used to drive data line D0 of the on-board data bus by using an 8089 tri-state driver. If a user program performs an INPUT from I/O port E30, data bit 0 will be set to 1 if the INTERRUPT REQUEST LATCH is set.

The purpose of INTERRUPT STATUS LATCH is to minimize the possibility of the asynchronous interrupt occuring while the interrupt status is being read by a bus master. If the latch was not included in the design and an asynchronous interrupt did occur while a bus master is reading MULTIBUS data line DAT0/, a data buffer on the master could go into a meta-stable state. By adding the extra latch, which is clocked by the IORD/ command for I/O port E30, the possibility of data line DAT0/ changing during a bus master read operation is eliminated.

The INTERRUPT REQUEST LATCH is cleared when a user program performs an OUTPUT to I/O port E30.

This interrupt structure assumes that several interrupt sources may exist on the same MULTI-BUS interrupt line (for example, INT3/). When the MULTIBUS master gets interrupted, it must poll the possible sources of the interrupt received and after determining the source of the interrupt, it must clear the INTERRUPT REQUEST LATCH for that particular interrupt source.

The interrupt logic for the 8-bit version of the design example is identical to the interrupt logic of the 8/16-bit version of the design example.

PPI Operation — Two 8255A Parallel Peripheral Interface (PPI) devices are shown interfaced to the slave design example logic. One PPI is connected to the on-board data bus lines D0 - D7 and is addressed with the even I/O port addresses E38, E3A, E3C, and E3E. The second PPI is connected to data bus lines D8 - DF and is addressed with the odd I/O port addresses E39, E3B, E3D, and E3F. The even or odd I/O port selection is controlled by using the ADR0 address line in the chip select term of the PPIs. In addition, the odd PPI (A11) is selected when the BHENBL term is high. This occurs when the MULTIBUS signal BHEN/ is low indicating that a word (16-bit) I/O instruction is being executed. When a word I/O instruction is executed, both PPIs will perform the I/O operation specified.

The specifications of the 8255A device state that the address lines A0 and A1 and the chip select lines must be stable before the $\overline{RD}$ or $\overline{WR}$ lines are activated. The MULTIBUS specification address set-up time of 50 ns and the short gate propagation delays in this design assure that the address lines are stable before $\overline{RD}$ or $\overline{WR}$ are active.

The data hold requirements of the 8255A were discussed in a previous section. The 8255A specification states that data will be stable on the data bus lines a maximum of 250 ns after a READ command. This specification was used to select the delay for the XACK/ signal.

The PPI operation for the 8-bit version of the design example is slightly different than that used for the 8/16-bit version. The chip select signal for the bottom PPI does not use the BHENBL term since 16-bit data transfers are not possible with an 8-bit I/O slave board. Also, the chip select and address signals have been swapped so the top PPI occupies I/O address range X8 - XB, and the bottom PPI occupies I/O address range XC - XF (X is the base address of the 8-bit version). This swapping of the address lines was not necessary; however, it was thought to be more convenient to access the PPIs in two groups of 4 contiguous I/O port addresses.

IV. SUMMARY

This application note has shown the structure of the Intel MULTIBUS system bus. The structure supports a wide range of system modules from the Intel OEM Microcomputer Systems product line that can be extended with the addition of user designed modules. Because the user designed modules are no doubt unique to particular applications, a goal of this application note has been to describe in detail the singular common element - the bus interface. Material has also been presented to assist the systems designer to understanding the bus functions so that successful systems integration can be achieved.

APPENDIX A

PIN ASSIGNMENT OF BUS SIGNALS ON MULTIBUS BOARD P1 CONNECTOR

	PIN	MNEMONIC	DESCRIPTION	PIN	MNEMONIC	DESCRIPTION
		(COMPONENT SIDE)			(CIRCUIT SIDE)	
POWER SUPPLIES	1	GND	Signal GND	2	GND	Signal GND
	3	+5V	+5Vdc	4	+5V	+5Vdc
	5	+5V	+5Vdc	6	+5V	+5Vdc
	7	+12V	+12Vdc	8	+12V	+12Vdc
	9	−5V	−5Vdc	10	−5V	−5Vdc
	11	GND	Signal GND	12	GND	Signal GND
BUS CONTROLS	13	BCLK/	Bus Clock	14	INIT/	Initialize
	15	BPRN/	Bus Pri. In	16	BPRO/	Bus Pri. Out
	17	BUSY/	Bus Busy	18	BREQ/	Bus Request
	19	MRDC/	Mem Read Cmd	20	MWTC/	Mem Write Cmd
	21	IORC/	I/O Read Cmd	22	IOWC/	I/O Write Cmd
	23	XACK/	XFER Acknowledge	24	INH1/	Inhibit 1 disable RAM
BUS CONTROLS AND ADDRESS	25		Reserved	26	INH2/	Inhibit 2 disable PROM or ROM
	27	BHEN/	Byte High Enable	28	AD10/	
	29	CBRQ/	Common Bus Request	30	AD11/	Address
	31	CCLK/	Constant Clk	32	AD12/	Bus
	33	INTA/	Intr Acknowledge	34	AD13/	
INTERRUPTS	35	INT6/	Parallel	36	INT7/	Parallel
	37	INT4/	Interrupt	38	INT5/	Interrupt
	39	INT2/	Requests	40	INT3/	Requests
	41	INT0/		42	INT1/	
ADDRESS	43	ADRE/		44	ADRF/	
	45	ADRC/		46	ADRD/	
	47	ADRA/	Address	48	ADRB/	Address
	49	ADR8/	Bus	50	ADR9/	Bus
	51	ADR6/		52	ADR7/	
	53	ADR4/		54	ADR5/	
	55	ADR2/		56	ADR3/	
	57	ADR0/		58	ADR1/	
DATA	59	DATE/		60	DATF/	
	61	DATC/		62	DATD/	
	63	DATA/	Data	64	DATB/	Data
	65	DAT8/	Bus	66	DAT9/	Bus
	67	DAT6/		68	DAT7/	
	69	DAT4/		70	DAT5/	
	71	DAT2/		72	DAT3/	
	73	DAT0/		74	DAT1/	
POWER SUPPLIES	75	GND	Signal GND	76	GND	Signal GND
	77		Reserved	78		Reserved
	79	−12V	−12Vdc	80	−12V	−12Vdc
	81	+5V	+5Vdc	82	+5V	+5Vdc
	83	+5V	+5Vdc	84	+5V	+5Vdc
	85	GND	Signal GND	86	GND	Signal GND
All Mnemonics © Intel Corporation 1978						

APPENDIX A (Continued)
P2 CONNECTOR PIN ASSIGNMENT OF OPTIONAL BUS SIGNALS

	(COMPONENT SIDE)			(CIRCUIT SIDE)	
PIN	MNEMONIC	DESCRIPTION	PIN	MNEMONIC	DESCRIPTION
1	GND	Signal GND	2	GND	Signal GND
3	5 VB	+5V Battery	4	5 VB	+5V Battery
5		Reserved	6	VCCPP	+5V Pulsed Power
7	–5 VB	–5V Battery	8	–5 VB	–5V Battery
9		Reserved	10	Reserved	
11	12 VB	+12V Battery	12	12 VB	+12V Battery
13	PFSR/	Power Fail Sense Reset	14	Reserved	
15	–12 VB	–12V Battery	16	–12 VB	–12V Battery
17	PFSN/	Power Fail Sense	18	ACLO	AC Low
19	PFIN/	Power Fail Interrupt	20	MPRO/	Memory Protect
21	GND	Signal GND	22	GND	Signal GND
23	+15V	+15V	24	+15V	+15V
25	–15V	–15V	26	–15V	–15V
27	PAR1/	Parity 1	28	HALT/	Bus Master HALT
29	PAR2/	Parity 2	30	WAIT/	Bus Master WAIT STATE
31			32	ALE	Bus Master ALE
33			34	Reserved	
35			36	Reserved	
37			38	AUX RESET/	Reset switch
39			40		
40			42		
43	Reserved		44		
45			46		
47			48		
49			50	Reserved	
51			52		
53			54		
55			56		
57			58		
59			60		

Notes:

1. PFIN, on slave modules, if possible, should have the option of connecting to INT0/ on P1.
2. All undefined pins are reserved for future use.

All Mnemonics © Intel Corporation 1978

APPENDIX B

BUS TIMING SPECIFICATIONS SUMMARY

Parameter	Description	Minimum	Maximum	Units
t_{BCY}	Bus Clock Period	100	D.C.	ns
t_{BW}	Bus Clock Width	$0.35\, t_{BCY}$	$0.65\, t_{BCY}$	
t_{SKEW}	BCLK/skew		3	ns
t_{PD}	Standard Bus Propagation Delay		3	
t_{AS}	Address Set-Up Time (at Slave Board)	50		ns
t_{DS}	Write Data Set Up Time	50		ns
t_{AH}	Address Hold Time	50		ns
t_{DHW}	Write Data Hold Time	50		ns
t_{DXL}	Read Data Set Up Time To XACK	0		ns
t_{DHR}	Read Data Hold Time	0	65	ns
t_{XAH}	Acknowledge Hold Time	0	65	ns
t_{XACK}	Acknowledge Time	0	t_{TOUT}	ns
t_{CMD}	Command Pulse Width	100	t_{TOUT}	ns
t_{ID}	Inhibit Delay	0	100 (Recommend < 100 ns)	ns
t_{XACKA}	Acknowledge Time of of an Inhibited Slave	t_{IAD} + 50 ns	t_{TOUT}	
t_{XACKB}	Acknowledge Time of an Inhibiting Slave	1.5	t_{TOUT}	μs
t_{IAD}	Acknowledge Disable from Inhibit (An internal parameter on an inhibited slave; used to determine t_{XACKA} Min.)	0	100 (arbitrary)	ns
t_{AIZ}	Address to Inhibits High delay		100	ns
t_{INTA}	INTA/ Width	250		ns
t_{CSEP}	Command Separation	100		ns

BUS TIMING SPECIFICATIONS SUMMARY

Parameter	Description	Minimum	Maximum	Units
t_{BREQL}	↓BCLK/ to BREQ/ Low Delay	0	35	ns
t_{BREQH}	↓BCLK/ to BREQ/ High Delay	0	35	ns
t_{BPRNS}	BPRN/ to ↓BCLK/ Setup Time	22		ns
t_{BUSY}	BUSY/ delay from ↓BCLK/	0	70	ns
t_{BUSYS}	BUSY/ to ↓BCLK/ Setup Time	25		ns
t_{BPRO}	↓BCLK/ to BPRO/ (CLK to Priority Out)	0	40	ns
t_{BPRNO}	BPRN/ to BPRO/ (Priority In to Out)	0	30	ns
t_{CBRO}	↓BCLK/ to CBRQ/ (CLKto Common Bus Request)	0	60	ns
t_{CBRQS}	CBRQ/ to ↓BCLK/ Setup Time	35		ns
t_{CPM}	Central Priority Module Resolution Delay (Parallel Priority)	0	$t_{BCY} - t_{BREQ}$ $-2t_{PD}$ $-t_{BPRNS}$ $-t_{SKEW}$	
t_{CCY}	C-clock Period	100	110	ns
t_{CW}	C-clock Width	$0.35\,t_{CCY}$	$0.65\,t_{CCY}$	ns
t_{INIT}	INIT/Width	5		ms
t_{INITS}	INIT/ to MPRO/ Setup Time	100		ns
t_{PBD}	Power Backup Logic Delay	0	200	ns
t_{PFINW}	PFIN/ Width	2.5		ms
t_{MPRO}	MPRO/ Delay	2.0	2.5	ms
t_{ACLOW}	ACLO/ Width	3.0		ms
t_{PFSRW}	PFSR/ Width	100		ns
t_{TOUT}	Timeout Delay	5	∞	ms
t_{DCH}	D.C. Power Supply Hold from ALCO/	3.0		ms
t_{DCS}	D.C. Power Supply Setup to ACLO/	5		ms

APPENDIX C
BUS DRIVERS, RECEIVERS, AND TERMINATIONS

Bus Signals	Driver 1,3 Location	Type	I_{OL} Min ma	I_{OH} Min μa	C_O Max pf	Receiver 2,3 Location	I_{IL} Max ma	I_{IH} Max μa	C_I Max pf	Termination Location	Type	R	Units
DAT0/→DATF/ (16 lines)	Masters and Slaves	TRI	16	-2000	300	Masters and Slaves	-0.8	125	18	1 place	Pullup	2.2	KΩ
ADR0/-ADRB/, BHEN/ (21 lines)	Masters	TRI	16	-2000	300	Slaves	-0.8	125	18	1 place	Pullup	2.2	KΩ
MRDC/,MWTC/	Masters	TRI	32	-2000	300	Slaves (Memory; memory-mapped I/O)	-2	125	18	1 place	Pullup	1	KΩ
IORC/,IOWC/	Masters	TRI	32	-2000	300	Slaves (I/O)	-2	125	18	1 place	Pullup	1	KΩ
XACK/	Slaves	TRI	32	-2000	300	Masters	-2	125	18	1 place	Pullup	510	Ω
INH1/,INH2/	Inhibiting Slaves	OC	16	—	300	Inhibited Slaves (RAM, PROM, ROM, Memory-Mapped I/O)	-2	50	18	1 place	Pullup	1	KΩ
BCLK/	1 place (Master us)	TTL	48	-3000	300	Master	-2	125	18	Mother-board	To +5V / To GND	220 / 330	Ω / Ω
BREQ/	Each Master	TTL	5	-400	60	Central Priority Module	2	50	18	Central Priority Module (not req)	Pullup	1	KΩ
BPRO/	Each Master	TTL	5	-400	60	Next Master in Serial Priority Chain at its BPRN/	-1.6	50	18	(not req)			
BPRN/	Parallel: Central Priority Module Serial:Prev Masters BPRO/	TTL	5	-400	300	Master	-2	50		(not req)			
BUSY/, CBRQ	All Masters	O.C.	32	—	300	All Masters	-2	50	18	1 place	Pullup	1	KΩ
INIT/	Master,	O.C.	32	—	300	All	-2	50	18	1 place	Pullup	2.2	KΩ
CCLK/	1 place	TTL	48	-3000	300	Any	-2	125	18	Mother-board	To +5V / To GND	220 / 330	Ω / Ω
INTA/	Masters	TRI	32	-2000	300	Slaves (Interrupting I/O)	-2	125	18	1 place	Pullup	1	KΩ
INT0/→INT7/ (8 lines)	Slaves	O.C.	16	—	300	Masters	-1.6	40	18	1 place	Pullup	1	KΩ
PFSR/	User's Fron Panel?	TTL	16	-400	300	Slaves, Masters	-1.6	40	18	1 place	Pullup	1	KΩ
PFSN/	Power Back Up Unit	TTL	16	-400	300	Masters	-1.6	40	16	1 place	Pullup	1	KΩ
ACLO	Power Supply	O.C.	16	-400	300	Slaves, Masters	-1.6	40	18	1 place	Pullup	1	KΩ
PFIN/	Power Back-Up Unit	O.C.	16	-400	300	Masters	-1.6	40	18	1 place	Pullup	1	KΩ
MPRO/	Power Back-Up Unit	TTL	16	-400	300	Slaves Masters	-1.6	40	18	1 place	Pullup	1	KΩ

BUS DRIVERS, RECEIVERS, AND TERMINATIONS

Bus Signals	Driver 1,3						Receiver 2,3				Termination			
	Location	Type	I_{OL} Min_{ma}	I_{OH} $Min_{\mu a}$	C_O Max_{pf}		Location	I_{IL} Max_{ma}	I_{IH} $Max_{\mu a}$	C_I Max_{pf}	Location	Type	R	Units
Aux Reset/	User's Front Panel?	Switch to GND	—	—	—		Masters	−2	50	18	None			

Notes:

1. Driver Requirements

   I_{OH} = High Output Current Drive
   I_{OL} = Low Output Current Drive
   C_O = Capacitance Drive Capability
   TRI = 3-State Drive
   O.C. = Open Collector Driver
   TTL = Totem-pole Driver

2. Receiver Requirements

   I_{IH} = High Input Current Load
   I_{IL} = Low Input Current Load
   C_I = Capacitive Load

3. TTL low state must be $\geq$ −0.5v but $\leq$ 0.8v at the receivers
   TTL high state must be $\geq$ 2.0v but $\leq$ 5.5v at the receivers

4. For the iSBC 80/10 and the iSBC 80/10A use only a 1K pull-up resistor to +5v for BCLK/ and CCLK/ termination.

BUS POWER SPECIFICATIONS

	Standard (P1)				Optional (P2)					
					Analog Power		Battery Power Backup			
	Ground	+ 5	+ 12	− 12	+ 15	− 15	+ 5	+ 12	− 12	− 5
Mnemonic	GND	+ 5V	+ 12V	− 12V	+ 15V	− 15V	+ 5B	+ 12B	− 12B	− 5B
Bus Pins	P1 + 1,2, 11,12, 75,76 85,86	P1 + 3,4, 5,6,81, 82,83, 84	P1 + 7,8	P1 + 79, 80	P2 + 23, 24	P2 + 25, 26	P2 + 3,4, 5,6	P2 + 11, 12	P2 + 15, 16	P2 − 7,8
Nominal Output	Ref.	+ 5.0V	+ 12.0V	− 12.0V	+ 15.0V	− 15.0V	+ 5.0V	+ 12.0V	− 12.0V	− 5.0V
Tolerance from Nominal[1]	Ref.	± 5%	± 5%	± 5%	± 3%	± 3%	± 5%	± 5%	± 5%	± 5%
Ripple (Pk–Pk)[2]	Ref.	50 mV	50 mV	50 mV	10 mV	10 mV	50 mV	50 mV	50 mV	50 mV
Transient Response Time[3]		500 μs	500μs	500 μs	100 μs	100 μs	500 μs	500 μs	500 μs	500 μs
Transient Deviation[4]		± 10%	± 10%	± 10%	± 10%	± 10%	± 10%	± 10%	± 10%	± 10%

NOTES:

1. Tolerance is worst case, including initial voltage setting line and load effects of power source, temperature drift, and any additional steady state influences.

2. As measured over any bandwidth not to exceed 0 to 500 kHz.

3. As measured from the start of a load change to the time an output recovers within ± 0.1% of final voltage.

4. Measured as the peak deviation from the initial voltage.

APPENDIX E
MECHANICAL SPECIFICATIONS

12.00 ±0.005

11.500

0.25 × 45°
2 PLACES

0.25

0.25

0.109 DIA
3 HOLES

4

COMPONENT SIDE

1

5.950
±0.005

6.20

6.75 REF

2

3

0.06R
TYP

0.55

0.30

3.080

0.390

6.767 ±0.005

4.570

CHAMFER ALL
CONNECTOR EDGES
0.040 × 45°

0.015 ± 0.005 × 45°
2 PLACES

NOTES:

1. BOARD THICKNESS: 0.062

2. MULTIBUS CONNECTOR: 86-PIN, 0.156 SPACING
   CDC VFB01E43D00A1
   VIKING 2VH43/1ANE5

3. AUXILIARY CONNECTOR: 60-PIN, 0.100 SPACING
   CDC VPB01B30D00A1
   TI H311130
   AMP PE5-14559

4. EJECTOR TYPE: SCANBE #S203

5. BUS DRIVERS AND RECEIVERS SHOULD BE LOCATED AS CLOSE AS POSSIBLE TO
   THEIR RESPECTIVE MULTIBUS PIN CONNECTIONS

6. BOARD SPACING: 0.6

7. COMPONENT HEIGHT: 0.4

8. CLEARANCE ON CONDUCTOR NEAR EDGES: 0.050

APPENDIX F
MULTIBUS™ SLAVE DESIGN EXAMPLE SCHEMATIC
8/16-BIT VERSION

MULTIBUS™ SLAVE DESIGN EXAMPLE SCHEMATIC 8/16-BIT VERSION

APPENDIX G
MULTIBUS™ SLAVE DESIGN EXAMPLE SCHEMATIC
8-BIT VERSION

MULTIBUS™ SLAVE DESIGN EXAMPLE SCHEMATIC 8-BIT VERSION

intel®

APPLICATION NOTE

AP-43

November 1978

Using The iSBC™ 957 Execution Vehicle For Executing 8086 Program Code

Joe Barthmaier
OEM Microcomputer Systems Applications

©Intel Corporation, 1978

9800816

1-79

Using The iSBC™ 957 Execution Vehicle For Executing 8086 Program Code

Contents

I. INTRODUCTION

The iSBC 957 Intellec—iSBC 86/12 Interface and Execution Package contains the hardware and software required to interface an iSBC 86/12 Single Board Computer with an Intellec Microcomputer Development System. The iSBC 957 package gives the 8086 user the capability to develop software on an Intellec System and then debug this software on an iSBC 86/12 board using a program download capability and an interactive system monitor. The 8086 user has all the capabilities of the Intellec system at his disposal and has the powerful iSBC 86/12 system monitor commands to use for debugging 8086 programs.

The iSBC 86/12 board is an Intel 8086 based processor board which, in addition to the processor, contains 32K bytes of dual port RAM, sockets for up to 16K bytes of ROM/EPROM, a serial I/O port, 24 parallel I/O lines, 2 programmable counters, 9 levels of vectored priority interrupts, and an interface to the MULTIBUS™ system bus. The iSBC 957 package consists of monitor EPROMs for the iSBC 86/12 board, Loader software for the Intellec system, four (4) cable assemblies, assorted line drivers and terminators, and signal adapters.

The iSBC 957 package provides the capability of downloading and uploading program and data blocks between an iSBC 86/12 board and an Intellec system. In addition, monitor commands and displays may be input and viewed from the Intellec system console. The iSBC 957 package, when used with the iSBC 86/12 board and an Intellec Microcomputer Development System, provides the user with the capability to edit, compile or assemble, link, locate, download, and interactively debug programs for the 8086 processor. The iSBC 957 package and the iSBC 86/12 board form an excellent "execution vehicle" for users developing software for the 8086 processor regardless of whether the users are 8086 component users or iSBC 86/12 board users. Using the iSBC 957 package 8086 programs may be debugged at the full 5 MHz speed of the processor. The recommended hardware for the execution vehicle is an iSBC 660 system chassis with an 8 card slot backplane and power supply, an iSBC 032 32K byte RAM memory board, the iSBC 957 package, and the iSBC 86/12 board.

This application note will describe how the iSBC 957 package may be used to develop and debug 8086 programs. First a description of the iSBC 86/12 board will be presented. Readers familiar with the iSBC 86/12 board may want to skip this section. Next follows a detailed description of the iSBC 957 package and the iSBC 86/12 system monitor commands. A program example of a matrix multiplication routine will then be presented. This example will contain both assembly language and PL/M-86 procedures. The steps required to compile, assemble, link, locate and debug the program code will be explained in detail. A typical debugging session using the iSBC 86/12 system monitor will be presented.

II. THE iSBC™ 86/12 SINGLE BOARD COMPUTER

The iSBC 86/12 Single Board Computer, which is a member of Intel's complete line of iSBC 80/86 computer products, is a complete computer system on a single printed-circuit assembly. The iSBC 86/12 board includes a 16-bit central processing unit (CPU), 32K bytes of dynamic RAM, a serial communications interface, three programmable parallel I/O ports, programmable timers, priority interrupt control, MULTIBUS control logic, and bus expansion drivers for interface with other MULTIBUS-compatible expansion boards. Also included is dual port control logic to allow the iSBC 86/12 board to act as a slave RAM device to other MULTIBUS masters in the system. Provision is made for user installation of up to 16K bytes of read only memory. Figure 1 contains a block diagram of the iSBC 86/12 board and in Appendix A is a simplified logic diagram of the iSBC 86/12 board.

Central Processing Unit

The central processor for the iSBC 86/12 board is Intel's 8086, a powerful 16-bit H-MOS device. The 225 sq. mil chip contains 29,000 transistors and has a clock rate of 5MHz. The architecture includes four (4) 16-bit byte addressable data registers, two (2) 16-bit memory base pointer registers and two (2) 16-bit index registers, all accessed by a total of 24 operand addressing modes for complex data handling and very flexible memory addressing.

Instruction Set — The 8086 instruction repertoire includes variable length instruction format (including double operand instructions), 8-bit and 16-bit signed and unsigned arithmetic operators for binary, BCD and unpacked ASCII data, and iterative word and byte string manipulation functions. The instruction set of the 8086 is a functional superset of the 8080A/8085A family and with
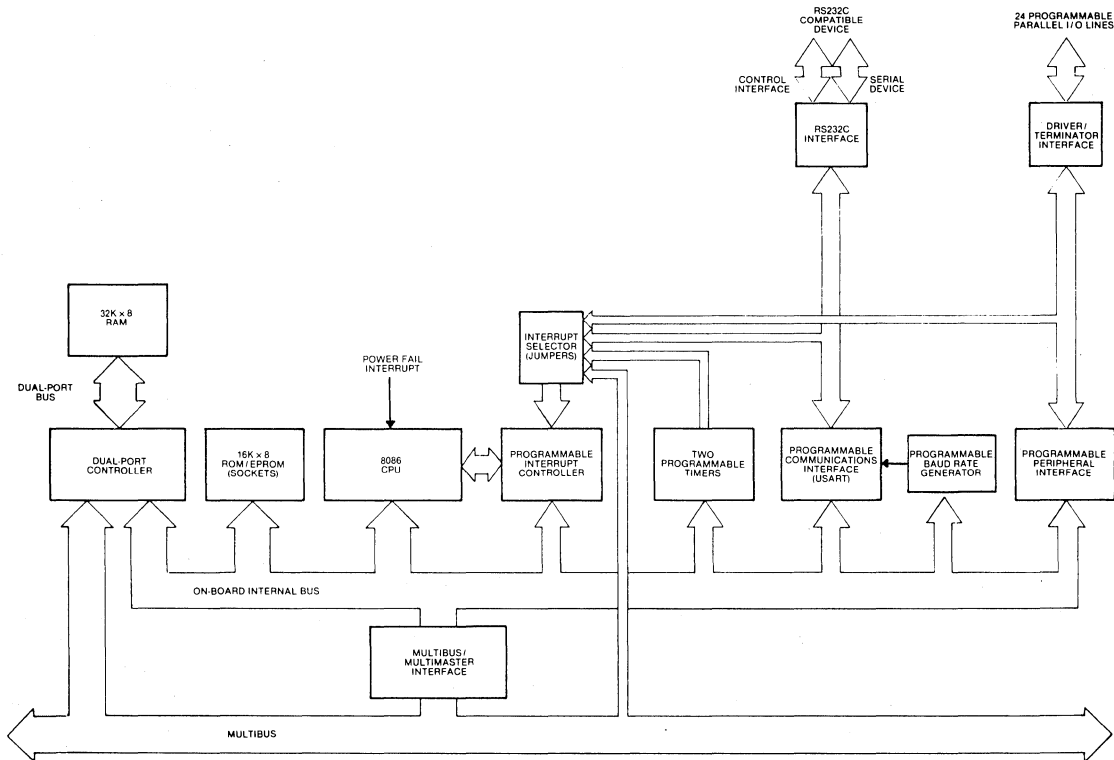
Figure 1. iSBC™ 86/12 Single Board Computer Block Diagram

available software tools, programs written for the 8080A/8085A can be easily converted and run on the 8086 processor.

Architectural Features — A 6-byte instruction queue provides pre-fetching of sequential instructions and can reduce the $1.2\,\mu$ sec minimum instruction cycle to 400 nsec by having the instruction already in the queue.

The stack oriented architecture facilitates nested sub-routines and co-routines, reentrant code and powerful interrupt handling. The memory expansion capabilities offer a 1 megabyte addressing range. The dynamic relocation scheme allows ease in segmentation of pure procedure and data for efficient memory utilization. Four segment registers (code, stack, data, extra) contain program loaded offset values which are used to map 16-bit addresses to 20-bit addresses. Each register maps 64K-bytes at a time and activation of a specific register is controlled explicitly by program control and is also selected implicitly by specific functions and instructions.

Bus Structure

The iSBC 86/12 board has an internal bus for communicating with on-board memory and I/O options, a system bus (the MULTIBUS) for referencing additional memory and I/O options, and the dual-port bus which allows access to RAM from the on-board CPU and the MULTIBUS System Bus. Local (on-board) accesses do not require MULTIBUS communication, making the system bus available for use by other MULTIBUS masters (i.e. DMA devices and other single board computers transferring to additional system memory). This feature allows true parallel processing in a multiprocessor environment. In addition, the MULTIBUS interface can be used for system expansion through the use of other 8- and 16-bit iSBC computers, memory and I/O expansion boards.

RAM Capabilities

The iSBC 86/12 board contains 32K-bytes of dynamic read/write memory. Power for the on-board RAM and refresh circuitry may be optionally provided on an auxiliary power bus, and

memory protect logic is included for RAM battery backup requirements. The iSBC 86/12 board contains a dual port controller which allows access to the on-board RAM from the iSBC 86/12's CPU and from any other MULTIBUS master via the system bus. The dual port controller allows 8- and 16-bit accesses from the MULTIBUS System Bus and the on-board CPU transfers data to RAM over a 16-bit data path. Priorities have been established such that memory refresh is guaranteed by the on-board refresh logic and that the on-board CPU has priority over MULTIBUS requests for access to RAM. The dual-port controller includes independent addressing logic for RAM access from the on-board CPU and from the MULTIBUS system bus. The on-board CPU will always access RAM starting at location 00000$_H$. Address jumpers allow on-board RAM to be located starting on any 8K-byte boundary within a 1 megabyte address range for accesses from the MULTIBUS system bus. In conjunction with this feature, the iSBC 86/12 board has the ability to protect on-board memory from MULTIBUS access to any contiguous 8K-byte segments. These features allow multi-processor systems to establish local memory for each processor and shared system (MULTIBUS) memory configurations where the total system memory size (including local on-board memory) can exceed 1 megabyte without addressing conflicts.

EPROM/ROM Capabilities

Four sockets are provided for up to 16K-bytes of non-volatile read only memory on the iSBC 86/12 board. Configuration jumpers allow read only memory to be installed in 2K, 4K, or 8K increments.

On-board ROM is accessed via 16 bit data paths. System memory size is easily expanded by the addition of MULTIBUS compatible memory boards available in the iSBC 80/86 family.

Parallel I/O Interface

The iSBC 86/12 board contains 24 programmable parallel I/O lines implemented using the Intel 8255A Programmable Peripheral Interface. The system software is used to configure the I/O lines in any combination of unidirectional input/output and bidirectional ports.

Therefore, the I/O interface may be customized to meet specific peripheral requirements. In order to take full advantage of the large number of possible I/O configurations, sockets are provided for interchangeable I/O line drivers and terminators. Hence, the flexibility of the I/O interface is further enhanced by the capability of selecting the appropriate combination of optional line drivers and terminators to provide the required sink current, polarity, and drive/termination characteristics for each application. The 24 programmable I/O lines and signal ground lines are brought out to a 50-pin edge connector that mates with flat, woven, or round cable.

Serial I/O

A programmable communications interface using the Intel 8251A Universal Synchronous/Asynchronous Receiver/Transmitter (USART) is contained on the iSBC 86/12 board. A software selectable baud rate generator provides the USART with all common communication frequencies. The USART can be programmed by the system software to select the desired asynchronous or synchronous serial data transmission technique (including IBM Bi-Sync). The mode of operation (i.e., synchronous or asynchronous), data format, control character format, parity, and baud rate are all under program control. The 8251A provides full duplex, double buffered transmit and receive capability. Parity, overrun, and framing error detection are all incorporated in the USART. The RS232C compatible interface on each board, in conjunction with the USART, provides a direct interface to RS232C compatible terminals, cassettes, and asynchronous and synchronous modems. The RS232C command lines, serial data lines, and signal ground line are brought out to a 26 pin edge connector that mates with RS232C compatible flat or round cable. The iSBC 530 teletypewriter adapter provides an optically isolated interface for those systems requiring a 20 mA current loop. The iSBC 530 adapter may be used to interface the iSBC 86/12 board to teletypewriters or other 20 mA current loop equipment.

Programmable Timers

The iSBC 86/12 board provides three independent, fully programmable 16-bit interval timers/event counters utilizing the Intel 8253 Programmable Interval Timer. Each counter is capable of operating in either BCD or binary modes. Two of these timers/counters are available to the systems designer to generate accurate time intervals under software control. Routing for the outputs and gate/trigger inputs of two of these counters is jumper selectable. The outputs may be independently routed to the 8259A Programmable Interrupt Controller and to the I/O line drivers associated with

the 8255A Programmable Peripheral Interface, or may be routed as inputs to the 8255A chip. The gate/trigger inputs may be routed to I/O terminators associated with the 8255A or as output connections from the 8255A. The third interval timer in the 8253 provides the programmable baud rate generator for the iSBC 86/12 RS232C USART serial port. In utilizing the iSBC 86/12, the systems designer simply configures, via software, each timer independently to meet system requirements. Whenever a given time delay or count is needed, software commands to the programmable timers/event counters select the desired function.

The contents of each counter may be read at any time during system operation with simple read operations for event counting applications, and special commands are included so that the contents can be ready "on the fly".

MULTIBUS™ and Multimaster Capabilities

The MULTIBUS system bus features asynchronous data transfers for the accommodation of devices with various transfer rates while maintaining maximum throughput. Twenty address lines and sixteen separate data lines eliminate the need for address/data multiplexing/demultiplexing logic used in other systems, and allow for data transfer rates up to 5 megawords/sec. A failsafe timer is included in the iSBC 86/12 board which can be used to generate an interrupt if an addressed device does not respond within 6 msec.

Multimaster Capabilities — The iSBC 86/12 board is a full computer on a single board with resources capable of supporting a great variety of OEM system requirements. For those applications requiring additional processing capacity and the benefits of multiprocessing (i.e., several CPUs and/or controllers logically sharing system tasks through communication over the system bus), the iSBC 86/12 board provides full MULTIBUS arbitration control logic. This control logic allows up to three iSBC 86/12 boards or other bus masters, including iSBC 80 family MULTIBUS compatible 8-bit single board computers, to share the system bus in serial (daisy chain) priority fashion, and up to 16 masters to share the MULTIBUS with the addition of an external priority network. The MULTIBUS arbitration logic operates synchronously with a MULTIBUS clock (provided by the iSBC 86/12 board or optionally provided directly from the MULTIBUS System Bus) while data is transferred via a handshake between the master and slave modules. This allows different speed controllers to share resources on the same bus, and transfers via the bus proceed asynchronously. Thus, transfer speed is dependent on transmitting and receiving devices only. This design prevents slow master modules from being handicapped in their attempts to gain control of the bus, but does not restrict the speed at which faster modules can transfer data via the same bus. The most obvious applications for the master-slave capabilities of the bus are multiprocessor configurations, high speed direct memory access (DMA) operations, and high speed peripheral control, but are by no means limited to these three.

Interrupt Capability

The iSBC 86/12 board provides 9 vectored interrupt levels. The highest level is the NMI (Non-Maskable Interrupt) line which is directly tied to the 8086 CPU. This interrupt cannot be inhibited by software and is typically used for signalling catastrophic events (e.g., power failure).

The Intel 8259A Programmable Interrupt Controller (PIC) provides vectoring for the next eight interrupt levels.

The PIC accepts interrupt requests from the programmable parallel and serial I/O interfaces, the programmable timers, the system bus, or directly from peripheral equipment. The PIC then determines which of the incoming requests is of the highest priority, determines whether this request is of higher priority than the level currently being serviced, and, if appropriate, issues an interrupt to the CPU. Any combination of interrupt levels may be masked, via software, by storing a single byte in the interrupt mask register of the PIC. The PIC generates a unique memory address for each interrupt level. These addresses contain unique instruction pointers and code segment offset values (for expanded memory operation) for each interrupt level. In systems requiring additional interrupt levels, slave 8259A PIC's may be interfaced via the MULTIBUS system bus, to generate additional vector addresses, yielding a total of 65 unique interrupt levels.

Interrupt Request Generation — Interrupt requests may originate from 16 sources. Two jumper selectable interrupt requests can be automatically generated by the programmable peripheral interface.

Two jumper selectable interrupt requests can be automatically generated by the USART when a character is ready to be transferred to the CPU or a character is ready to be transmitted.

A jumper selectable request can also be generated by each of the programmable timers. Eight additional interrupt request lines are available to the user for direct interface to user designated peripheral devices via the system bus, and two interrupt request lines may be jumper routed directly from peripherals via the parallel I/O driver/terminator section.

Power-Fail Control

Control logic is also included to accept a power-fail interrupt in conjunction with the AC-low signal from the iSBC 635 Power Supply or equivalent.

Expansion Capabilities

Memory and I/O capacity may be expanded and additional functions added using Intel MULTIBUS compatible expansion boards. High speed integer and floating point arithmetic capabilities may be added by using the iSBC 310 high speed mathematics unit. Memory may be expanded to 1 megabyte by adding user specified combinations of RAM boards, EPROM boards, or combination boards. Input/output capacity may be increased by adding digital I/O and analog I/O expansion boards. Mass storage capability may be achieved by adding single or double density diskette controllers. Modular expandable backplanes and cardcages are available to support multiboard systems.

III. THE iSBC™ 957 PACKAGE

The iSBC 957 Intellec—iSBC 86/12 Interface and Execution Package extends the software development capabilities of the Intellec Microcomputer Development systems to the Intel 8086 CPU. Programs for the 8086 may be written in PL/M-86 and/or assembly language and compiled or assembled on the Intellec system. These programs may then be downloaded from an Intellec ISIS-II disk file to the iSBC 86/12 board for execution and debug. The programs will execute at the full 5 MHz clock rate of the 8086 CPU with no speed degradation caused by the iSBC 957 hardware or software. Special communication software allows transparent access to the powerful interactive debug commands in the iSBC 86/12 monitor from the Intellec console terminal. These debug commands include single-step instruction execution, execution with breakpoints, memory and register displays, memory searches, comparison of two memory blocks and several other commands. After a debugging session, the debugged program code may be uploaded from the iSBC 86/12 board to an Intellec ISIS-II disk file.

The iSBC 957 Intellec—iSBC 86/12 Interface and Execution Package consists of the following:

a. Four Intel 2716 EPROMs which contain the system monitor program for the iSBC 86/12 board.

b. An ISIS-II diskette containing loader software for execution in the Intellec which provides for communications between the user or an Intellec ISIS-II file and the iSBC 86/12 board. Also included on the diskette are a library of routines for system console I/O.

c. Four cable assemblies used for transmitting commands, code and data between the iSBC 86/12 board and the Intellec system.

d. An iSBC 530 adapter assembly which converts serial communications signals from current loop to RS232C.

e. Line drivers and terminators used for the iSBC 86/12 parallel ports.

f. A small printed circuit board which is plugged into an iSBC 86/12 receiver/terminator socket and is used when program code is downloaded or uploaded using the parallel cable.

iSBC™—Intellec™ Configurations

There are two distinct functional configurations for the iSBC 957 package; one configuration for the Intellec Series II, Models 220 or 230 development systems and another for the Intellec 800 series development systems.

Intellec Series II System Configurations

When used with Intellec Series II Model 220 or 230 systems, a set of cables are used to connect the serial I/O port edge connector on the iSBC 86/12 board and the SERIAL 1 output port on the Intellec system. This configuration is shown in Figure 2. How this system functions is explained in the following paragraphs.

The SERIAL 1 port on the Intellec Series II Model 220 or 230 system is an RS232 port which is designed for use with a data terminal. This port may be used on the Intellec system for interfacing to RS232 devices such as CRT terminals or printers. The serial ports on the iSBC 86/12 board and the Intellec systems are connected as shown in the Figure 2. The flat ribbon cable connected to the iSBC 86/12 board has an edge connector for connecting to the board on one end and a standard RS232 connector on the other end. The second cable, the RS232 Up/Down Load cable, has an RS232 connector on each end. This cable, however,
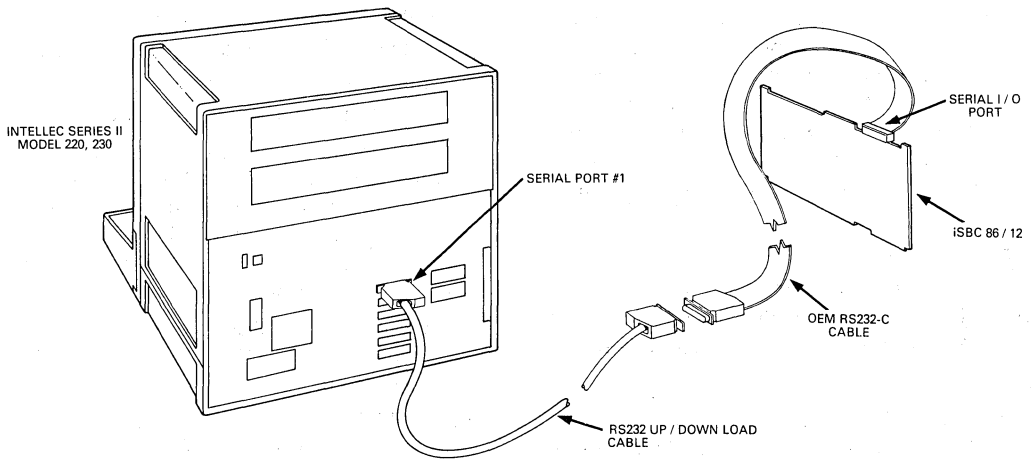
Figure 2. Intellec™ Series II Model 220, 230—iSBC™ 86/12 Configuration

is not a standard cable with the RS232 signals bussed between identically numbered pins on each of the connectors. The schematic for the cable is shown in Figure 3. Note that the TXD (transmit data) and the RXD (receive data) and the RTS (ready to send) and the CTS (clear to send) signals have been crossed. This is done because both the Intellec system and the iSBC 86/12 board are configured to act as data sets which are communicating with data terminals. Swapping these signals permits the units to communicate directly with no modifications to the Intellec or iSBC 86/12 systems themselves.
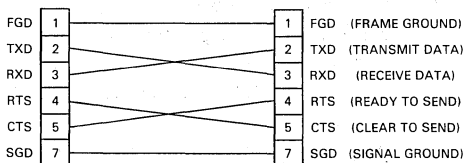
Figure 3. Intellec™—iSBC™ 86/12 RS232 UP/DOWN LOAD Cable

The software in the Intellec system accepts characters output from the iSBC 86/12 board through the Intellec SERIAL 1 port. The software then outputs these characters on the CRT monitor built into the Intellec Series II Model 220 or 230. In a similar fashion, characters input from the Intellec key-board are passed down the serial link to the iSBC 86/12 monitor program. The integrated CRT monitor and keyboard on the Intellec system then becomes the "virtual terminal" of the iSBC 86/12 monitor program. If this were the only function of the iSBC 957 package, there would be no real benefit to the user. However, when the "virtual terminal" capability is combined with the capability to download and upload program code and data files between the Intellec ISIS-II file system and the iSBC 86/12 board, a very powerful software development tool is realized. The software in the Intellec system must examine the commands which are input from the keyboard and in the case of the LOAD and TRANSFER commands (see later sections for details on monitor commands), the software must open and read or write ISIS-II disk files.

Transfer rates using Intellec Series II Model 220 or 230 system are 9600 baud when transferring hexadecimal object files to or from a disk file and 600 baud when transferring commands between the iSBC 86/12 board and the CRT monitor and keyboard. With a 9600 baud transfer rate, it is possible to load 64K bytes of memory in about four minutes.

Intellec 800 System Configurations

The iSBC 957 package may be used with the Intellec 800 system in four different configurations. These four configurations are determined by two

variables. The first variable is whether the iSBC 86/12 board is connected to the Intellec 800 TTY port or to the Intellec 800 CRT port. The second variable is whether or not a parallel cable is used for uploading and downloading hexadecimal object files. Figures 4A and 4B illustrate the four configurations.

In Figure 4A, the configuration shows the TTY port of the Intellec 800 system connected to the iSBC 86/12 serial port using two cables and an iSBC 530 teletypewriter adapter. The TTY port of the Intellec 800 system is designed for using a teletypewriter as the Intellec console device. To use this port for communication with the iSBC 86/12 board, the current loop TTY signal must be converted to an RS232 compatible voltage signal. This function is performed by the iSBC 530 adapter.

The cable which connects the Intellec 800 system to the iSBC 530 adapter performs a function similar to the RS232 Up/Down Load cable described above. A schematic for this cable and all other components of the iSBC 957 package are included with the delivered product.

The transfer rate for both commands and data when the TTY port is connected to the iSBC 86/12 board is 110 baud. This means to download even moderately sized programs would require large amounts of time, several minutes or even hours. However, much faster times may be achieved by using the parallel ports of the iSBC 86/12 board and the Intellec system for downloading program files. This parallel port used on the Intellec 800 system is the output port labeled PROM which is normally used with the Universal Prom Pro-
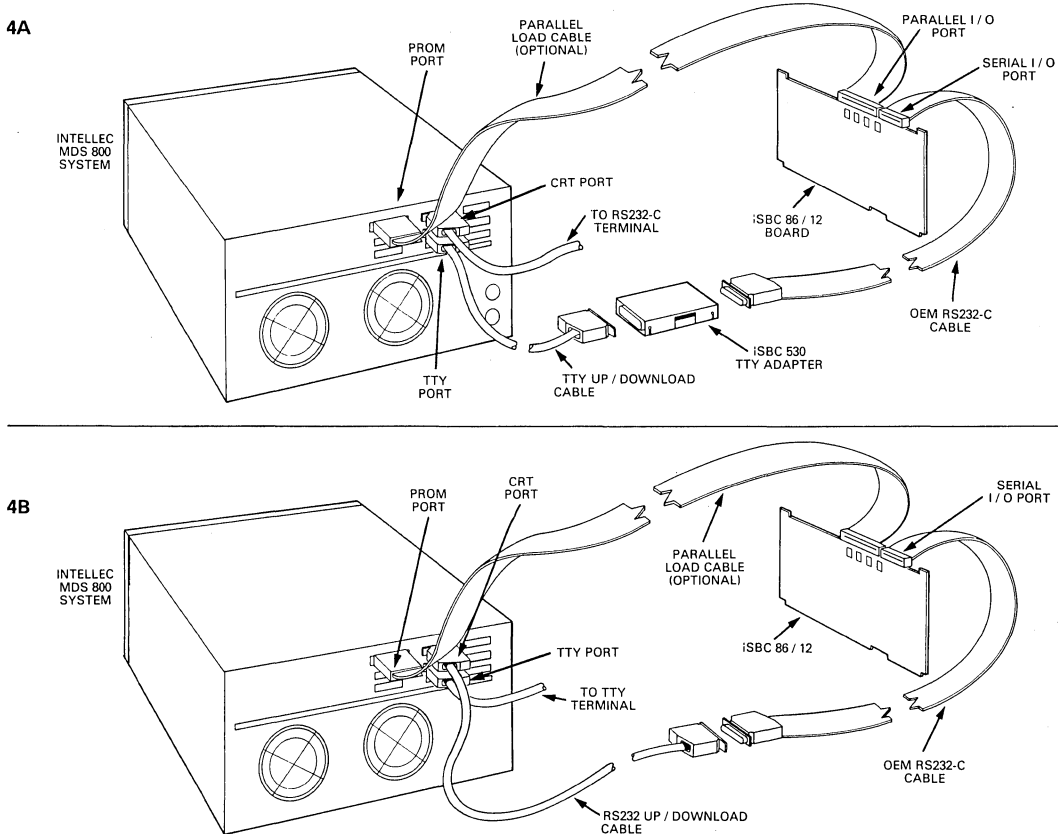
Figure 4A, 4B. Intellec™ 800—iSBC™ 86/12 Configurations

grammer. A cable is connected between the Intellec PROM port and the parallel I/O port, J1 of the iSBC 86/12 board. Parallel port B of the iSBC 86/12 board is used for the 8-bit byte transfers from the Intellec system to the iSBC 86/12 board, port A is used for the byte transfers from the iSBC 86/12 board to the Intellec system and port C is used for controlling the byte transfers. A special status adapter piggyback board must be inserted into a receiver/terminator socket of the iSBC 86/12 board. This status adapter circuit is required to provide the necessary handshaking signals from the iSBC 86/12 parallel ports to the Intellec PROM port.

The transfer rate achieved when downloading and uploading hexadecimal object files with the parallel cable is approximately 1,000 bytes per second. The time required to load 64K bytes of memory is approximately 2½ minutes.

Figure 4B shows a configuration with the Intellec 800 CRT port connected to the serial port of the iSBC 86/12 board. The TTY port of the Intellec 800 system is connected to a teletypewriter or some other current loop device to act as a system console. The optional parallel load cable is also shown. The cables used for this configuration are the same as those used with the Intellec Series II Configurations. Command transfer rates require 110 baud because the TTY port of the Intellec 800 system is used for communicating with the console device. However, hexadecimal object files can be loaded at 9600 baud since this operation uses only the Intellec to iSBC 86/12 RS232 link.

It is also possible to download files with the parallel cable, this mode being somewhat faster than the serial download mode (2½ minutes versus four minutes for 64K bytes of memory). Table I contains a summary of the command and memory transfer rates for each of the Intellec-iSBC 86/12 configurations.

Comparing the Intellec 800 configurations shown in Table 1 and in Figures 4A and 4B it should be noted:

1. Using the TTY port (Figure 4A) of the Intellec 800 system for communications with the iSBC 86/12 board (essentially) requires installation of the parallel cable and jumper modifications for downloading and uploading files, and thus, prevents the use of the parallel ports for other I/O functions.

2. Using the CRT port (Figure 4B) of the Intellec

800 system for communication with the iSBC 86/12 board provides for a fast serial download capability, thus freeing the parallel ports for other uses. However, this configuration requires a teletypewriter or a CRT capable of accepting a current loop input signal as the Intellec system console.

Table 1
COMMAND AND MEMORY TRANSFER RATES FOR INTELLEC—iSBC™ 86/12 CONFIGURATIONS

	INTELLEC SERIES II 220/230 SERIAL PORT TO iSBC 86/12	INTELLEC 800 TTY PORT TO iSBC 86/12	INTELLEC 800 CRT PORT TO iSBC 86/12
Effective Command Rate	600 Baud	110 Baud	110 Baud*
Load/Transfer Rate			
Serial	9600 Baud	110 Baud	9600 Baud
Parallel	N/A	1K bytes/sec**	1K bytes/sec**
Approximate Time to load 64K bytes of memory			
Serial	4 minutes	5 hours	4 minutes
Parallel	N/A	2.5 minutes	2.5 minutes

*The actual baud rate of the Intellec—iSBC 86/12 link is 9600 baud, but the effective command rate is determined by the slower Intellec—console serial link.

**Transmission rate over the parallel link is determined by the speed of the two processors and is approximately 1K bytes per second.

IV. THE iSBC 957—iSBC 86/12 MONITOR PROGRAM

The iSBC 86/12 monitor program is an EPROM resident program which facilitates debugging of user written programs. The monitor program used in the iSBC 86/12 board with the iSBC 957 package is the same monitor program written to interface the iSBC 86/12 directly to an RS232C data terminal. When interfaced directly to a terminal, the iSBC 86/12 board functions in a stand-alone environment communicating directly with the user via the data terminal. A user may use the monitor for entering small programs in hexadecimal format, executing a program, examining registers and memory contents, etc.

To use the monitor program with an Intellec system, the proper cables must be installed and the iSBC 957 Loader program must be loaded into Intellec memory and executed. The Loader program is resident on a file named SBC861, and when executed, the Loader outputs a sign-on message. Next, the iSBC 86/12 monitor program must be started and the baud rate of the iSBC 86/12 to Intellec serial communications link must be determined. This is done by pressing the RESET switch on the chassis

Table 2
MONITOR COMMAND LIST

COMMAND	FUNCTION AND SYNTAX
L Load Hex Object File	Loads hexadecimal object file from Intellec into iSBC 86/12 memory using serial (S) or parallel (P) mode. L{S|P},< filename>[,<bias addr>]<cr>
T Transfer Hex Object File	Transfers blocks of iSBC 86/12 memory to Intellec as a hex object file using serial (S) or parallel (P) mode. T[X] {S|P} ,<start addr>,<end addr>,<filename> [,<exec addr>]<cr>
E Exit	Exits the loader program and returns to ISIS. E<cr>
N Single Step	Executes one user program instruction. N[<addr>],[[<addr>],[*<cr>
G Go	Transfers control of the 8086 CPU to the user program with up to 2 optional breakpoints. G[<start addr>][,<break 1 addr> [,<break 2 addr>]]<cr>
S Substitute Memory	Displays/modifies memory locations in byte or word format. S[W]<addr>,[[new contents],]*<cr>
X Examine/Modify Register	Displays/modifies 8086 CPU registers. X[<reg>][[<new contents>],]*<cr>
D Display Memory	Displays contents of a memory block in byte or word format. D[W]<start addr>[,<end addr>]<cr>
M Move	Moves contents of a memory block. M<start addr>,<end addr>,<destination addr><cr>
C Compare	Compares two memory blocks. C<start addr>,<end addr>,<destination addr><cr>
F Find	Searches a memory block for a byte or word constant. F[W]<start addr>,< end addr>,<data><cr>
H Hex Arithmetic	Performs hexadecimal addition and subtraction. H<data 1>,<data 2><cr>
I Port Input	Inputs and displays byte or word data from input port. I[W]<port addr>,[,]*<cr>
O Port Output	Outputs byte or word data to output port. O[W]<port addr>,<data>[,<data>]*<cr>

Syntax conventions used in the command structure are as follows:

[A]   indicates that "A" is optional

[A]* indicates one or more optional iterations of "A"

<B>   indicates that "B" is a variable

{A\|B} indicates "A" or "B"

<cr> indicates a carriage return is entered

Numeric arguments can be expressed as a number, the contents of a register, or the sum or difference of numbers and register contents. Thus, addresses and data can be expressed as follows:

    addr ::=   [<expr>:]<expr>

    expr ::=   <number>|<register>|<expr> {+ | −} <number>|
                <expr> {+ | −} <register>

    register ::= AX|BX|CX|DX|SP|BP|SI|DI|CS|DS|SS|ES|IP|FL

    number ::= <digit>|<digit><number>

    digit ::=   0|1|2|3|4|5|6|7|8|9|A|B|C|D|E|F

Numeric fields within arguments are entered as hexadecimal numbers. The valid range of numerical values is from 0000-FFFF. Larger numbers may be entered, but only the last four digits (or two in the case of byte values) are significant. Leading zeros may be omitted.

An address argument consists of a segment value and an offset value separated by a colon (:). If a segment value is not specified, the default segment value is the CS register value.

containing the iSBC 86/12 board and typing two "U"s on the Intellec console. The ASCII uppercase character U has a binary pattern of alternating ones and zeros, the iSBC 86/12 monitor uses this pattern to determine the baud rate of the serial link. After the baud rate has been determined, the monitor program outputs a sign-on message to the console. An example of loader program execution and monitor program initialization is shown below (user entered characters are underlined).

:F1:SBC861
ISIS-II iSBC 86/12 LOADER, Vx.x
(user resets iSBC 86/12 board and types two "U"s)
iSBC 86/12 MONITOR, Vy.y
.

The monitor prompts with a period "." when it is ready for a command. The user can then enter a command file, which consists of a one- or two-character command followed by zero, one, or more arguments. The command may be separated from the first argument by an optional single space; a single comma is required as a delimiter between arguments. The command line is terminated by a carriage return or a comma depending on the command, and no action takes place until the command terminator is sensed. The user can cancel a command before entering the command terminator by pressing any illegal key (e.g., rubout or Control-X).

Table 2 contains a summary of the loader and monitor commands. These commands will not be explained in detail; instead, the next section of the application note will show examples of using these loader and monitor commands. The iSBC 957 User's Guide referenced at the front of this document does, however, contain a complete description of each of the monitor and loader commands.

Table 3 contains a list of the 8086 hardware registers and abbreviations used by the monitor program.

Table 3
8086 CPU REGISTERS

REGISTER NAME	ABBREVIATION
Accumulator	AX
Base	BX
Count	CX
Data	DX
Stack Pointer	SP
Base Pointer	BP
Source Index	SI
Destination Index	DI
Code Segment	CS
Data Segment	DS
Stack Segment	SS
Extra Segment	ES
Instruction Pointer	IP
Flag	FL

Figure 5 memory map labels:

Left side:
- ON-BOARD EPROM (8K bytes): FFFFF$_H$ — FE000$_H$ — MONITOR PROGRAM
- 8000$_H$ — AVAILABLE USER AREA
- ON-BOARD RAM (32K bytes): 1C0$_H$ INITIAL USER STACK, 130$_H$ MONITOR DATA AREA, A0$_H$, INTERRUPT VECTORS 0-39, 0$_H$

Right side (8259A PIC VECTORS):

39	INTR 7	9C$_H$
38	INTR 6	98$_H$
37	INTR 5	94$_H$
36	INTR 4	90$_H$
35	INTR 3	8C$_H$
34	INTR 2	88$_H$
33	INTR 1	84$_H$
32	INTR 0	80$_H$
31		
	RESERVED FOR FUTURE USE BY INTEL	
5		
4	Interrupt on Overflow	10$_H$
3	One-Byte Intr Instruction	C$_H$
2	Non-Maskable Intr	8$_H$
1	Single Step	4$_H$
0	Divide by Zero	0$_H$

Figure 5. Memory Map of iSBC™ 86/12 Memory With Monitor Program

Figure 5 contains a memory map of the iSBC 86/12 memory with the monitor program. Note that the monitor uses the top 8K bytes of memory for its program code and the first 384 bytes of memory (locations $\emptyset$ hex to 17F hex) for monitor and user stack, data and interrupt vectors. When the monitor program is reset, the segment registers, the IP and the flags are set to $\emptyset$; and the SP is set to $\emptyset$1C$\emptyset$H allowing 64 bytes for the user's stack. If 64 bytes is not sufficient for the user's application program, the SP should be set to some other value. The monitor program sets the single-step, one-byte instruction trap and non-maskable interrupt vectors to monitor entry points. The monitor also sets the 8259A Priority Interrupt Controller to fully nested mode with level $\emptyset$ at the highest priority and all interrupts unmasked. The eight interrupt vector addresses for the 8259A are also set to addresses in the monitor. User programs may change the 8259A interrupt vectors to interrupt service routine addresses within the user programs; it is not necessary for users to program the 8259A chip directly. When an interrupt occurs, control passes to either the monitor or directly to user code depending on the address stored in the vector location. When the monitor responds to an interrupt, it acknowledges the interrupt and displays the interrupt level, CS and IP register values and next instruction byte on the system console (e.g., I = 3 @ 100:230F F5).

When a user requests a breakpoint with a "G" command, the monitor inserts the single byte instruction trap instructions (INT 3) in the location where the breakpoint is requested. It is also possible for the user to code an INT 3 instruction in his program. When a user coded INT 3 instruction is executed, the monitor will be re-entered and a line with the format @<CS Value>:<IP Value> <Instruction byte> will be displayed (e.g., @1200:3FO2 F5).

Included on the diskette with the Loader program are two libraries containing I/O routines for the console. The library files are named SBCIOS.LIB and SBCIOL.LIB; they contain similar routines. The routines in SBCIOS.LIB are written to be called with intrasegment subroutine calls, a PL/M-86 module compiled with the "small" control generates this type of call. The routines in SBCIOL.LIB are written to be called with intersegment subroutine calls, a PL/M-86 module compiled with either the "medium" or "large" control generates this type of call.

The console input output routines, CI and CO, contained in the library should be used when performing character input and output on the console. Example PL/M-86 calls to the two routines are:

```
CI: PROCEDURE BYTE EXTERNAL;
    END CI;
CO: PROCEDURE (X) EXTERNAL;
    DECLARE X BYTE;
    END CO;

          .
          .
          .
DECLARE INPUT$CHAR,
        OUTPUT$CHAR BYTE;
          .
          .
          .
INPUT$CHAR = CI;
          .
          .
          .
CALL CO(OUTPUT$CHAR);
          .
          .
          .
```

General Comments on Use of the iSBC 957 Package

1. If the iSBC 86/12 board is reset any time after the initial baud rate search, it is not necessary to reload the iSBC 957 Loader program or to download the program code a second time to the iSBC 86/12 board. It is only necessary to re-establish the communications link by typing two "U"s for the baud rate search.

2. The iSBC 86/12 board should not be plugged into an available card slot in an Intellec chassis; a separate chassis should be used. There are at least three reasons for this:

   a. There is only one RESET signal available on the Intellec system bus. Thus, each processor may not be reset independently. This means that the iSBC 86/12 board cannot be reset without re-booting the ISIS-II operating system and restarting the iSBC 957 Loader.

   b. The Intellec system uses five of the eight available interrupts on the system bus. This severely restricts the range of interrupts available to the iSBC 86/12 board. Also, the iSBC 86/12 board cannot turn-off the interrupt lamps on the Intellec front panel.

   c. The iSBC 86/12 board may address up to 1 Megabyte of memory using a 20 bit address. Many Intellec systems contain boards which generate and decode only the low order 16 address bits. For example, the iSBC 016 memory expansion board and the Intellec 800

monitor PROMs only decode 16 address bits. Memory expansion above 64K bytes in these systems is difficult since the boards which decode only 16 bits will force "holes" in the address space above 64K.

3. The iSBC 86/12 board is delivered with two inputs to the 8259A Priority Interrupt Controller connected. Interrupt request 2 (IR2) is connected to the counter $\emptyset$ output of the 8253 Programmable Interval Timer. IR5 is connected to the INT5/signal of the MULTIBUS System Bus. If these interrupts are not desired, the wire wrap jumpers making the connections should be removed from the iSBC 86/12 board. A particular problem may exist with the counter $\emptyset$ connection to IR2. If the 8253 counter $\emptyset$ is not specifically initialized with software, a low frequency square wave output will exist at counter $\emptyset$'s output. This may cause unwanted interrupts when interrupts are enabled by user programs.

4. If the iSBC 86/12 board is used in a system with expansion boards, it is important that the MULTIBUS bus exchange pins be properly jumpered. For example, if the iSBC 86/12 board is used with an iSBC 032 expansion memory board in a system, the BPRN/ MULTIBUS pin for the iSBC 86/12 board should be grounded.

   In addition, if any interrupts are used with the iSBC 86/12 board the BPRN/ pin must be grounded. This is true in both single and multiple board systems.

5. Certain user systems require more than one single board computer in the system for performing the functions required by the application. The MULTIBUS System Bus has been specifically designed to permit multiple CPU boards to communicate and to share system resources. However, debugging systems with multiple CPUs has always posed somewhat of a problem. The iSBC 957 package provides a solution to this problem. The serial cable which connects the iSBC 86/12 board to the Intellec system may be removed after the program has been downloaded to the iSBC 86/12 board. A console CRT may then be connected directly to the iSBC 86/12 board and the monitor program may be used to debug the program running on the board. Other iSBC 86/12 boards may also be downloaded from the Intellec system and then switched to their own local terminals. An 8-bit processor board, such as the iSBC 80/30 board, may also be included

in the system and ICE-85™ may be used for debugging the iSBC 80/30 program concurrently with the iSBC 86/12 programs. Using this scheme, it is possible to debug a system which has several CPU boards by setting breakpoints and using other debugging features on each of the individual CPUs.

V. MATRIX MULTIPLICATION EXAMPLE

To illustrate how the iSBC 957 package can be used to assist in the writing and debugging of 8086 programs on the iSBC 86/12 board, an example program of a matrix multiplication will be presented. The example chosen has been intentionally kept simple and straightforward. The emphasis of this section will be to document the steps required to assemble, compile, link, locate and debug software using an Intellec system, the iSBC 957 package and the iSBC 86/12 board. Part of the example will be written in 8086 assembly language and part in PL/M-86.

The main program is written in PL/M-86. The main program first performs some initialization and the matrix multiplication, then the program calls an assembly language procedure (subroutine), a PL/M-86 procedure and the console output procedure CO supplied in the I/O library on the iSBC 957 diskette. A flow diagram for the example program is shown in Figure 6.

Explanation of the Program Code

The program code is contained in three software modules EXECUTION$VEHICLE, FIND, and SBCCO. EXECUTION$VEHICLE contains the main program coded in PL/M-86 and the binary to ASCII conversion procedure BINDECASC also coded in PL/M-86. The module FIND contains the assembly language procedure FIND$MX which searches a matrix for its maximum value. The module SBCCO resides in the library of console I/O routines supplied with the iSBC 957 package. The procedure CO will be used from this library.

The program code for the EXECUTION$VEHICLE and FIND modules will be explained in the following paragraphs. Appendix B contains compilation and assembly listings for the two modules; also contained in Appendix B is a memory and debug map for the linked modules. The listings contain circled reference letters (e.g., Ⓐ) which are referred to by the code description below. The listings in the appendix have been printed on fold-out pages so that they may easily be seen when reading the text.
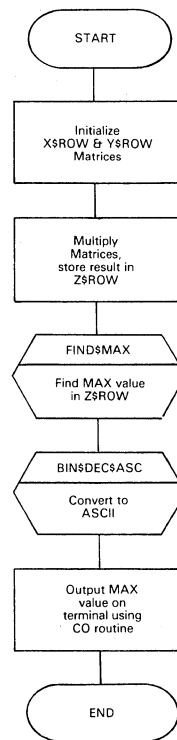
Figure 6.
Flow Diagram of Matrix Multiplication Example

Much of the description given below assumes that the reader is familiar with the PL/M-86 language and compiler, the 8086 assembler, and the link and locate program QRL86. It is recommended that the reader have at least a cursory knowledge of these subjects. The Intel literature for these subjects is listed near the front of this application note.

The EXECUTION$VEHICLE Module

Ⓐ The first section of the module includes introductory comments and then statements to declare the matrices, other variables, and procedures used in the program. Note that the matrix dimensions are declared using the literals M, N, and P which are initially set to 6, 5, and 3. Later in this note, other values for M, N, and P will be used.

Ⓑ The next section of code contains the statements which initialize the two matrices that will be multiplied X$ROW and Y$ROW.

As a result of this initialization, the two matrices will contain values as shown in Figure 7.

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 & 2 \\ 3 & 3 & 3 & 3 & 3 \\ 4 & 4 & 4 & 4 & 4 \\ 5 & 5 & 5 & 5 & 5 \end{bmatrix} \qquad \begin{bmatrix} 0 & -1 & -2 \\ 0 & -1 & -2 \\ 0 & -1 & -2 \\ 0 & -1 & -2 \\ 0 & -1 & -2 \end{bmatrix}$$

X\$ROW (6X5)   **Y\$ROW (5X3)**

Figure 7.
X\$ROW and Y\$ROW Matrices After Initialization

(C) The next program section performs the matrix multiplication. The algorithm required to multiply two matrices X and Y, storing the result in a third matrix Z is:

$$Z_{mp} = \sum_{i=1}^{n} X_{mi} {}^{*} Y_{ip}$$

Assuming X to be 6X5 matrix and Y a 5X3 matrix then

$$Z_{11} = X_{11}Y_{11} + X_{12}Y_{21} + X_{13}Y_{31} + X_{14}X_{41} + X_{15}Y_{51}$$

Thus, the upper left term is equal to the sum of the products of the top row of the X matrix times the left column of the Y matrix. The result that is obtained by multiplying the two matrices X\$ROW and Y\$ROW after they are initialized as explained above, is shown in Figure 8.

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & -5 & -10 \\ 0 & -10 & -20 \\ 0 & -15 & -30 \\ 0 & -20 & -40 \\ 0 & -25 & -50 \end{bmatrix}$$

Z\$ROW (6X3)

Figure 8. Result of Multiplying the Initialized Matrices X\$ROW and Y\$ROW

(D) The external assembly language procedure FIND\$MX is called to determine the maximum value in the matrix. The procedure is a typed procedure and returns the maximum value to the calling program which stores it in the integer variable MAX.

(E) The maximum value is then converted to a six (6) digit ASCII character string by the procedure BIN\$DEC\$ASC. The character string is stored in the array MAX\$ASC\$ARRAY, which contains the sign of the number and five (5) digits for the magnitude.

(F) Finally, the characters "MAX VALUE =" are output on the system console followed by the 6 ASCII characters containing the maximum value. The PL/M-86 built-in procedure SIZE returns the number of bytes of the array TEXT as a word value. The PL/M-86 built-in procedure SIGNED changes the type of the value from WORD to INTEGER. This is required so that the type of the arguments in the DO statement agree. The console output procedure CO is used to output the characters on the system console.

(G) Also contained in the module MATRIX.PLM is the binary to ASCII conversion procedure BIN\$DEC\$ASC. The first portion of the code contains the comments explaining the parameters and the calling sequence followed by the declarations. Note that the address of the array where the characters are to be stored is passed to the procedure and that the characters will be stored in the array using based variables. The next section of the code stores either a + or − sign in the first character position of the ASCII array and stores the absolute value of VALUE in the variable TEMP. Finally, the binary value is converted to ASCII using the algorithm explained in the comments. The MOD operator returns the remainder of the division by 10. The UNSIGN built-in procedure is required to change the type of the expression from INTEGER to WORD.

The FIND Module

(H) The FIND module contains the assembly language procedure FINDMX. The calling sequence and the parameters are explained in the comments at the beginning of the listing. Note that the label FINDMX has been declared PUBLIC so the link program can fill in its address in the CALL statement in the main program of module EXECUTION\$VEHICLE.

(I) The FIND module will contain three segments: a data segment, a stack segment and a code segment. It will be both convenient and pragmatic to append these three segments to the code, data and stack segments created by the

compiler for the EXECUTION$VEHICLE module. To accomplish this, the three segments must be given the same SEGMENT and CLASS names as those given these segments by the compiler. The SEGMENT and CLASS names used by the compiler are CODE, DATA, and STACK. The GROUP statements are used to place the segments DATA and STACK in the group DGROUP and the segment CODE in the group CGROUP. These group definitions conform with the group definitions generated by the PL/M-86 compiler when the SMALL size control option is used. A group is a collection of segments which requires less than 64K bytes of memory.

The ASSUME directive informs the assembler that the DS and SS registers will contain the base address of DGROUP and the CS register will contain the base address of CGROUP. This information will be used by the assembler when constructing machine instructions.

(J) The first segment appearing in the module is the data segment. The order of the segments is arbitrary, although it is recommended that the data segment precede the code segment to minimize forward references to variables which may cause the assembler to generate longer instruction codes. The data segment is declared PUBLIC, aligned on a WORD boundary and given both a segment and class name of DATA. Then follows the contents of the segment. In this particular example, only one word of storage is required. The ENDS directive indicates the end of the segment.

(K) Next comes the stack segment which is given the segment name of STACK, the combine-type attribute of STACK and the class name of STACK. The combine-type attribute of STACK assures that the stack storage required in this module will be appended to the storage required in the PL/M-86 compiled modules. Two bytes of stack are required by the code in this module, however, the monitor uses 13 words of stack when breakpoints and interrupts are used. Therefore, 14 words are reserved for the stack.

(L) Finally comes the code segment. The code segment has been given a segment name and class name of CODE and a group name of CGROUP, and has been declared PUBLIC. The alignment attribute of BYTE is specified

since it is desired that the code from this module be appended directly to the code from other modules without gaps between the code modules.

The assembly language code follows next. The code for the procedure must be enclosed between a pair of PROC, ENDP statements. The PROC statement is given the label FINDMX and specified as a NEAR procedure indicating it will be called with a near (intra-segment) CALL instruction and not a far (inter-segment) CALL instruction.

The comments at the beginning of the module and adjacent to the program statements explain the function being performed by the assembly language code.

The SBCCO Module

(M) The console output procedure CO is contained in the object module SBCCO of the library file SBCIOS.LIB. SBCIOS.LIB is part of the iSBC 957 package I/O libraries. The calling sequence and parameters for CO may be seen in the external procedure declaration in the EXECUTION$VEHICLE module.

Compiling the EXECUTION$VEHICLE Module

The EXECUTION$VEHICLE module is stored on a file named MATRIX.PLM on disk device :F1:. To compile the module, the following command line is used:

– PLM86 :F1:MATRIX.PLM DEBUG

This command line will cause the module stored in the file :F1:MATRIX.PLM to be compiled. The object code generated will be stored in a file with the default name :F1:MATRIX.OBJ and the listing generated will be stored in a file with the default name :F1:MATRIX.LST. To override the default object and listing files, the NOOBJECT and NOLIST compiler control switches can be used. File names for the listing and object files may also be specified in the command line. The DEBUG compiler control switch causes the compiler to generate extra symbol and line number information which will be used during debugging of the program. A listing of the compiled EXECUTION$VEHICLE module is contained in Appendix B.

To aid in the debugging of the program, the module was compiled a second time with the following command line:

– PLM86 :F1:MATRIX.PLM NOOBJECT
CODE DEBUG PRINT (:F1:MATRIX.XLS)

This command line specified that no object file is to be created and a listing file should be stored in the file :F1:MATRIX.XLS. The CODE compiler control switch causes the compiler to list the assembly language statements which the compiler has generated for each line of PL/M code. The listing stored in the file MATRIX.XLS is contained in Appendix C.

Assembly of the FIND Module

The assembly language module FIND is stored on a file named FIND.ASM, to assemble this module the following command line is used:

ASM86 :F1:FIND.ASM DEBUG

This command line will cause the FIND module to be assembled with the object code stored in the default file :F1:FIND.OBJ and the listing stored in the default file :F1:FIND.LST. The listing of the assembled FIND module is contained in Appendix B.

Linking and Locating the Object Module

To link and locate the object modules, the QRL86 program will be used. The QRL86 program performs both the linking and the locating of the object modules in a single step. QRL86 is primarily designed for the debugging stages of program development. Some applications may require the extended capabilities of the separate LINK and LOCATE programs when the final link and locate is performed. The command line used to invoke the QRL86 program is:

QRL86 :F1:MATRIX.OBJ, :F1:FIND.OBJ,
SBCIOS.LIB ORIGIN (1000H)

This command line will cause QRL86 to link the code from the three modules and to locate the resultant absolute object module starting at location 1000 hexadecimal. The iSBC 86/12 monitor uses the first 180H bytes of memory for the monitor stack, data and interrupt vectors, 1000H was chosen as a convenient starting address for the program. The absolute object code will be stored in a default file :F1:MATRIX (note no file name extension is used). By default, the memory and debug maps which are generated are stored in the file :F1:MATRIX.MPQ and are contained in Appendix B.

(N) The memory map contains the starting addresses and sizes of the CODE, CONST, DATA, STACK and MEMORY segments of the object module. Note that the start address

for the program is specified as ($0100H$, $0002H$) indicating a CS value of $0100H$ and an IP value of $0002H$ or an absolute value of $01002H$. The first two bytes of the code segment contain address values which the code generated by the compiler will use for setting up the DS and SS registers. The memory map shows the code segments from the three modules collected into the group CGROUP. The code segment from the EXECUTION$VEHICLE module is given the segment and class names of CODE and is put into CGROUP by the PL/M compiler. To assure that the code segment from the FIND module is concatenated with the code segment from the EXECUTION$VEHICLE module the identical class, segment and group names were specified in the SEGMENT and GROUP statements in the FIND module. Next, the group DGROUP is shown in the memory map. DGROUP contains 4 segments labelled CONST, DATA, STACK and MEMORY. Putting all of these segments in the same group tells the linker that they will all be in the same 64K block of memory. The SMALL size control option of the compiler, which was invoked by default, creates CGROUP, DGROUP, and the segments contained in them.

(P) The debug map contains the memory address of variables, instruction labels and the addresses of each code line of the PL/M-86 module. Notice that the variable storage labels have their addresses specified in the format (DS register value, displacement). For example, the variable TEMP has an address of DS=$012AH$, displacement = $000CH$ or an absolute address of $0136H$. Instruction labels and line numbers use the format (CS register value, IP register value). Thus, line number six (6) in the module EXECUTION$VEHICLE has the address CS=$0100H$, IP=$0B5H$ or $011B5H$.

Object to Hex Conversion

Before downloading the program to the iSBC 86/12, the format of the object module must be converted from the absolute object module format which QRL86 creates to a hexadecimal/ASCII representation of the object module. This is done using the program OH86 with the following command line:

OH86 :F1:MATRIX TO :F1:MATRIX.HEX

Downloading and Debugging the Program

The hardware configuration used for debugging the matrix multiplication example program code was

an Intellec Series II Model 230 development system, the iSBC 957 package, an iSBC 86/12 board, and an iSBC 660 system chassis. What follows is the system-user dialog for a typical debugging session.

The first step required is to bootstrap load the ISIS-II operating system by hitting the RESET switch of the Intellec. The Intellec resident loader software is then loaded and executed. Throughout the dialog which follows operator entered characters will be underlined:

```
ISIS-II, V3.4
-SBC861

ISIS-II ISBC 86/12 LOADER, V1.2
.
```

To initialize the iSBC 86/12 monitor, the user must hit the RESET switch on the iSBC 660 chassis and type two "U"s on the system console. The monitor program will output a line on the console when it is properly initialized.

```
 ISBC 86/12 MONITOR, V1.2
```

The monitor command "X" is typed to check that the monitor is properly operating and to examine the contents of the 8086 registers.

```
.X
AX=0000 BX=0000 CX=0000 DX=0000 SP=01C0 BP=0000 SI=0000
DI=0000 CS=0000 DS=0000 SS=0000 ES=0000 IP=0000 FL=0000
.
```

To download the hex object file to the iSBC 86/12, the "L" command is used. Because an Intellec Series II Model 230 is being used, a serial download is specified. The hex file name is MATRIX.HEX which is resident on disk device :F1:.

```
.LS,:F1:MATRIX.HEX
```

The "X" command is used again to examine the CPU registers. Note that the monitor has changed the contents of the CS and IP registers to the value of the starting address of the program.

```
.X
AX=0000 BX=0000 CX=0000 DX=0000 SP=01C0 BP=0000 SI=0000
DI=0000 CS=0100 DS=0000 SS=0000 ES=0000 IP=0002 FL=0000
.
```

The "D" command is next used to display the first 101 bytes of the program code. Unless another segment register is specified, the display command assumes all addresses specified are relative to the CS register. Thus, the code displayed will be from absolute addresses 1000 through 1100. The program code displayed may be compared with program code generated by the PL/M-86 compiler shown in Appendix C, code line 36.

```
.D0,100
0000 2A 01 FA 2E 8E 16 00 00 BC D0 00 8B EC 16 1F FB
0010 C7 06 8E 00 00 00 81 3E 8E 00 05 00 7E 03 E9 3C
0020 00 C7 06 90 00 00 00 81 3E 90 00 04 00 7E 03 E9
0030 22 00 8B 06 3E 00 B9 0A 00 F7 E9 8B 36 90 00 D1
0040 E6 89 C3 8B 0E 8E 00 89 88 10 00 81 06 90 00 01
0050 00 E9 D3 FF 81 06 8E 00 01 00 E9 B9 FF C7 06 8E
0060 00 00 00 81 3E 8E 00 04 00 7E 03 E9 40 00 C7 06
0070 90 00 00 00 81 3E 90 00 02 00 7E 03 E9 26 00 8B
0080 06 90 00 F7 D8 50 8B 06 3E 00 B9 06 00 F7 E9 8B
0090 36 90 00 D1 E6 89 C3 59 89 88 4C 00 81 06 90 00
00A0 01 00 E9 CF FF 81 06 8E 00 01 00 E9 B5 FF C7 06
00B0 92 00 00 00 81 3E 92 00 02 00 7E 03 E9 8C 00 C7
00C0 06 8E 00 00 00 81 3E 8E 00 05 00 7E 03 E9 72 00
00D0 8B 06 8E 00 B9 06 00 F7 E9 8B 36 92 00 D1 E6 89
00E0 C3 C7 80 6A 00 00 00 C7 06 90 00 00 00 81 3E 90
00F0 00 04 00 7E 03 E9 41 00 8B 06 8E 00 B9 0A 00 F7
0100 E9
.
```

The PL/M-86 compiler ends the main program in the EXECUTION$VEHICLE module with a halt instruction. After execution of the program it is more desirable to return to the monitor. To accomplish this, an INT 3 instruction (code=CC) will be substituted for the halt instruction (code=F4) at the address of 1B4H relative to a CS value of 100H. First the "D" command is used to verify the address of the halt instruction, then the "S" command is used to change the instruction to an INT 3 instruction.

```
.D1B4
01B4 F4
.S1B4, F4- CC
.
```

To execute the PL/M-86 main program, the "G" command is used. After the "G" is typed, the current contents of the IP are output, followed by the contents of the byte pointed to by the IP. A new value for the IP or breakpoint addresses may be specified before a carriage return <CR> is typed. In this example, only a <CR> is typed.

```
.G 0002- FA
MAX VALUE = -00050
@0100:01B5 55
.
```

The program executes and outputs the maximum value of the matrix calculated. The INT 3 instruction is executed which causes a return to the monitor. The monitor types out an at-sign (@) followed by the CS and IP register values and the first byte of the instruction following the INT 3 instruction.

The "X" command is typed to examine the CPU registers. Note that the program has set both the SS and DS registers to 012A. (012A0H is the address of the DGROUP as shown in the memory map.)

```
.X
AX=0030 BX=0005 CX=000A DX=0000 SP=00D0 BP=00D0 SI=0001
DI=0006 CS=0100 DS=012A SS=012A ES=0000 IP=01B5 FL=F202
.
```

The three matrices are displayed. Note that a word

display has been specified by using the "DW" Command and that the addresses have been specified relative to the DS register. The addresses of XROW, YROW, and Z$ROW may be found in the debug map given by QRL86. Note that the values stored in the matrices are the same as those shown in Figures 8 and 9.

```
.DW DS:10,4A
0010 0000 0000 0000 0000 0000 0001 0001 0001
0020 0001 0001 0002 0002 0002 0002 0002 0003
0030 0003 0003 0003 0003 0004 0004 0004 0004
0040 0004 0005 0005 0005 0005 0005
.DW DS:4C,68
004C 0000 FFFF
0050 FFFF 0000 FFFF FFFE 0000 FFFF FFFE 0000
0060 FFFF FFFE 0000 FFFF FFFE
.DW DS:6A,8C
006A 0000 0000 0000
0070 0000 FFFB FFF6 0000 FFF6 FFEC 0000 FFF1
0080 FFE2 0000 FFEC FFD8 0000 FFE7 FFCE
.
```

The "G" Command is used to reset the IP register to the start address of the program (002) and to specify a breakpoint at address 0AEH, which is the address of statement 57 of the main program. Statement 57 is the point in the program after the X$ROW and Y$ROW matrices have been initialized, but before the matrix multiplication is performed. After the <CR> is typed, the program executes until the breakpoint is encountered. At this point, the monitor outputs a line specifying the number of the breakpoint, the CS and IP values and the first byte of the next instruction to be executed.

```
.G 01B5- 55 002,AE

BR1 @0100:00AE C7
.
```

Next, the single-step capability is used with the "N" command to execute single instructions. At any time, CPU registers may be examined or changed. In this example, the "X" command is used. Execution of succeeding instructions is caused by typing a comma (,).

```
.N 00AE- C7 ,
00B4- 81 ,
00BA- 7E ,
00BF- C7
.X
AX=0018 BX=0018 CX=FFFE DX=0000 SP=00D0 BP=00D0 SI=0004
DI=0006 CS=0100 DS=012A SS=012A ES=0000 IP=00BF FL=F293
.N 00BF- C7 ,
00C5- 81 ,
00CB- 7E
.
```

The contents of the X$ROW and Y$ROW matrices are examined and changed with the "SW" (substitute word) command. If a comma (,) is typed after the contents of memory are displayed, then the contents are left unchanged and the next word of memory is displayed. If a value followed by a comma or <CR> is entered, then the contents are changed. If a <CR> is entered, the substitute

sequence is terminated.

```
.SW DS:1A, 0001- ,
001C 0001- ,
001E 0001- 10
.SW DS:5A, FFFF- ,
005C FFFE- ,
005E 0000- ,
0060 FFFF- 64
.
```

After the matrices are modified, execution is resumed with the "G" command. The max value is output and the INT 3 instruction executed. Finally, the contents of the 3 matrices are displayed.

```
.G 00CB- 7E
MAX VALUE = +00400
@0100:01B5 55
.DW DS:10,8C
0010 0000 0000 0000 0000 0000 0001 0001 0010
0020 0001 0001 0002 0002 0002 0002 0002 0003
0030 0003 0003 0003 0003 0004 0004 0004 0004
0040 0004 0005 0005 0005 0005 0005 0000 FFFF
0050 FFFE 0000 FFFF FFFE 0000 FFFF FFFE 0000
0060 0064 FFFE 0000 FFFF FFFE 0000 0000 0000
0070 0000 0051 FFD8 0000 00C0 FFEC 0000 0120
0080 FFE2 0000 0180 FFD8 0000 01E0 FFCE
.
```

Expanding the Example Program's Memory Requirements

To illustrate how the iSBC 86/12 board may be used for executing 8086 programs which require large amounts of RAM, the example program will be modified. The matrix dimensions of the example will be changed from values of 6, 5 and 3 for the literal symbols of M, N, and P to values of 100, 50, 70. The three matrices will then be of size 100X50, 50X70, and 100X70. The memory required for these matrices is 15.5K words or 31K bytes. The data, constant, stack and memory segments which are contained in the group DGROUP will now comprise almost 32K bytes of memory.

The extra memory requirements will be supplied by using an iSBC 032 board with the iSBC 86/12 board in the iSBC 660 chassis. The iSBC 032 board is a 32K byte RAM board which is compatible with both 8- and 16-bit CPU boards. The base address of the board may be selected anywhere in a 0 to 1 megabyte range on any 16K byte boundary. 8- or 16-bit data transfers may be selected. The iSBC 032 board will be jumpered to respond to addresses in the 512K or 544K address space (20 bit hex address range to 80000H to 87FFFH). This will illustrate the capabilities of the 8086 to access a 20-bit, 1 megabyte address range.

One other modification is required to the program. The magnitude of the numbers which would result from multiplying matrices of this size would greatly exceed the capacity of the 16-bit integer storage, even with the two matrices initialized to the small

values they presently contain. To keep the example simple, the initialization values will be changed so all elements of the X$ROW matrix are set equal to 2 and all elements of the Y$ROW matrix are set equal to 3. The result of the multiplication should make all the elements of Z$ROW equal to 300.

The modified lines of program code are shown below.

```
                    /* MATRIX DIMENSIONS */
27   1     DECLARE M LITERALLY '100';
28   1     DECLARE N LITERALLY '50';
29   1     DECLARE P LITERALLY '70';


36   1     DO I = 0 TO (M-1);
37   2        DO J = 0 TO (N-1);
38   3           X$ROW(I).COL(J) = 2;
39   3        END;
40   2     END;


41   1     DO I = 0 TO (N-1);
42   2        DO J = 0 TO (P-1);
43   3           Y$ROW(I).COL(J) = 3;
44   3        END;
45   2     END;
```

The EXECUTION$VEHICLE module must be re-compiled and then the three program modules must be linked and located using the QRL86 program. Specifying the SEGMENTS option of QRL86, the origin of the CODE segment which is in the group CGROUP is set at 1000H, as in the first example. However, the origin of the CONST, DATA STACK and MEMORY segments which make up the group DGROUP is set at 80000H.

> QRL86 :F1:MATRIX.OBJ,:F1:FIND.OBJ,
> SBCIOS.LIB SEGMENTS (CODE(1000H),
> CONST (80000H), DATA STACK, MEMORY)

The memory map generated by QRL86 shows the CGROUP having a start address of 01000H and the DGROUP having a start address of 80000H.

```
INVOKED BY:
QRL86 :F1:MATRIY.OBJ,:F1:FIND.OBJ,SBCIOS.LIB &
SEGMENTS(CODE(1000H),CONST(80000H),DATA,STACK,MEMORY)

INPUT MODULES INCLUDED:
:F1:MATRIY.OBJ(EXECUTIONVEHICLE)
:F1:FIND.OBJ(FIND)
SBCIOS.LIB(SBCCO)

RESULT WRITTEN TO :F1:MATRIY(EXECUTIONVEHICLE)
START ADDRESS IS (0100H,0002H)

 START   LTH   ALIGN  NAME                      CLASS

 01000H  298H   G    /GS/ CGROUP
 01000H  21DH   W    CODE(EXECUTIONVEHICLE)     CODE
 0121DH   41H   B    CODE(FIND)                 CODE
 0125EH   3AH   W    CODE(SBCCO)                CODE
                     /GE/ CGROUP
 80000H 7970H   G    /GS/ DGROUP
 80000H    CH   W    CONST(EXECUTIONVEHICLE)    CONST
 8000CH    0H   W    CONST(SBCCO)               CONST
 8000CH 792AH   W    DATA(EXECUTIONVEHICLE)     DATA
 87936H    2H   W    DATA(FIND)                 DATA
 87938H    0H   W    DATA(SBCCO)                DATA
 87940H   30H   SW   STACK                      STACK
 87970H    0H   W    MEMORY                     MEMORY
                     /GE/ DGROUP
 87970H    0H   G    ??SEG(FIND)                (NULL)
```

The object code is then converted to hex format and downloaded to the iSBC 86/12 board. When the program is executed, the maximum value is calculated and output on the console.

```
-SBC861

ISIS-II ISBC 86/12 LOADER, V1.2

ISBC 86/12 MONITOR,,V1.2
.LS,:F1:MATRIY.HEX

.S1AC, F4- CC
.G 0002- FA
MAX VALUE = +00300
@0100:01AD 55
.
```

VI. CONCLUSION

This application note has described the iSBC 957 Intellec—iSBC 86/12 Interface and Execution Package, and how this package may be used to develop and debug programs for the 8086 processor. First, the iSBC 86/12 single board computer was described, followed by a detailed description of the iSBC 957 package and the iSBC 86/12 system monitor commands. The power and versatility of the iSBC 957 package and monitor commands for developing and debugging programs for the 8086 were illustrated by a program example. In the example a program which consisted of PL/M-86 and assembly language routines was presented. The program code was explained, and the steps required to compile, assemble, link, locate, and debug the program were illustrated. Finally, a typical de-bugging session using the iSBC 86/12 system monitor which illustrates the powerful capabilities of the monitor was presented.
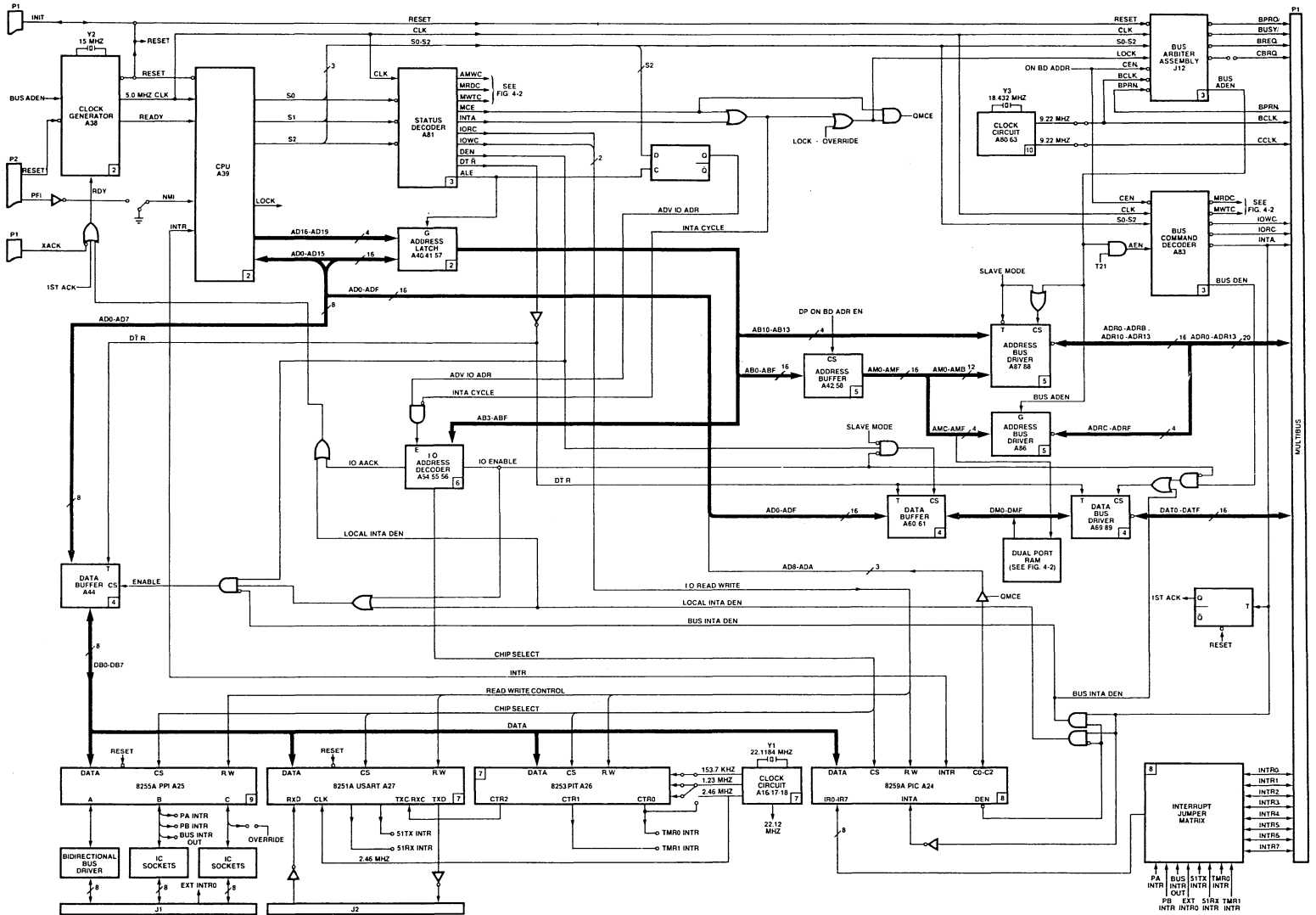
iSBC™ 86/12 SIMPLIFIED LOGIC DIAGRAM
INPUT / OUTPUT AND INTERRUPT
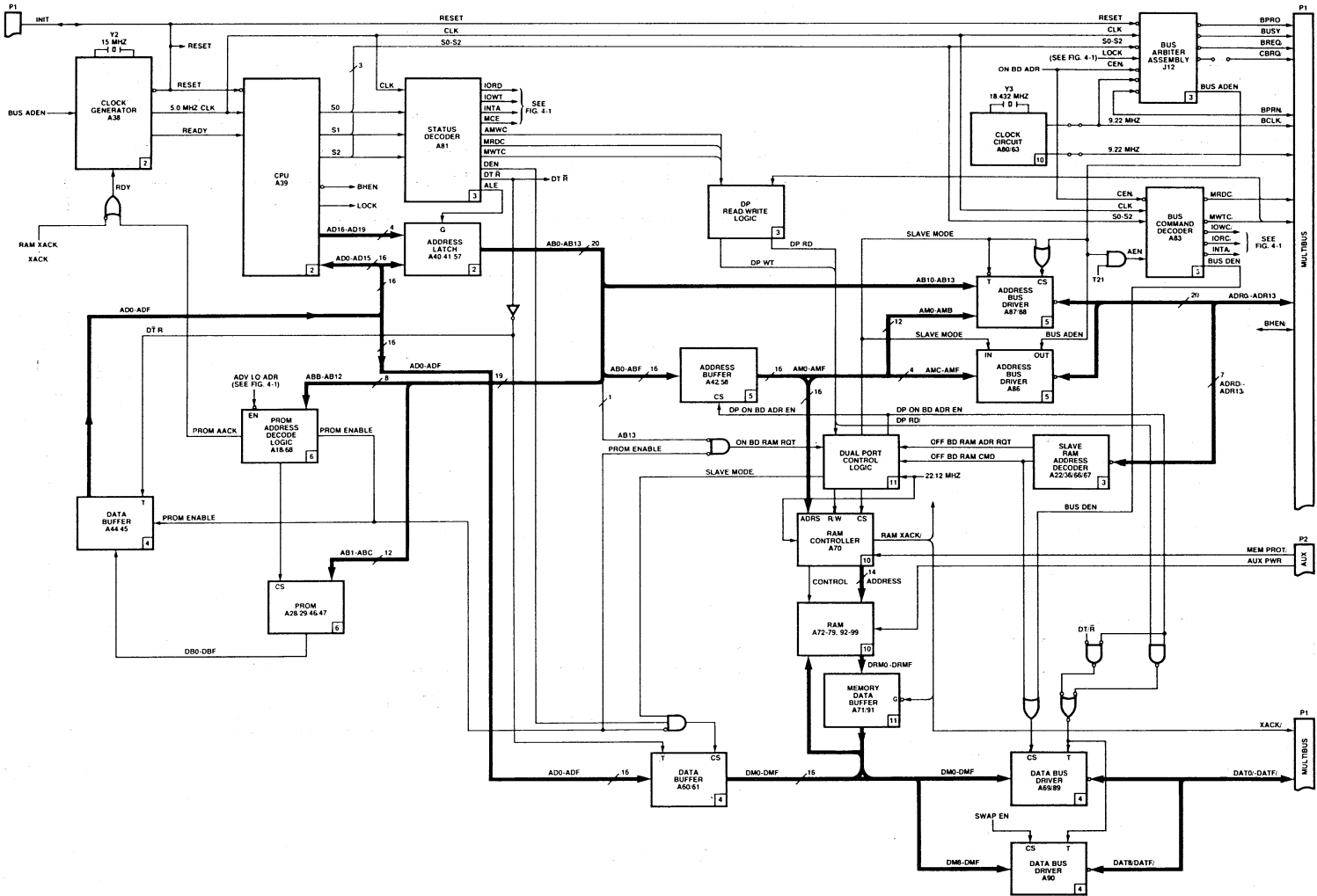
iSBC™ 86/12 SIMPLIFIED LOGIC DIAGRAM
ROM/EPROM AND DUAL PORT RAM

APPENDIX B

PROGRAM LISTINGS FOR EXECUTION$VEHICLE AND FIND MODULES

```
ISIS-II PL/M-86 V1.0 COMPILATION OF MODULE EXECUTIONVEHICLE
OBJECT MODULE PLACED IN :F1:MATRIX.OBJ
COMPILER INVOKED BY:  PLM86 :F1:MATRIX.PLM DEBUG


          /*      MATRIX MULTIPLICATION EXAMPLE PROGRAM

             PL/M-86 MAIN PROGRAM WHICH:
                  A) INITIALIZES TWO INTEGER MATRICES
                  B) MULTIPLIES THE TWO MATRICES AND STORES THE RESULT IN A
                     THIRD MATRIX
                  C) CALLS AN ASSEMBLY LANGUAGE PROCEDURE WHICH SEARCHES THE
                     THIRD MATRIX FOR THE MAXIMUM VALUE
                  D) CALLS A PL/M PROCEDURE WHICH CONVERTS THE MAXIMUM VALUE
                     FROM INTEGER TO ASCII
                  E) CALLS A PROCEDURE WHICH OUTPUTS THE ASCII CHARACTERS ON
                     THE SYSTEM CONSOLE
          */
 1        EXECUTION$VEHICLE:
          DO;


          /*  FIND$MX - EXTERNAL ASSEMBLY LANGUAGE PROCEDURE WHICH SEARCHES A
                  MATRIX FOR THE LARGEST ABSOLUTE MAGNITUDE.
                  PARAMETERS:
                      MATRIX$ADR -  ADDRESS OF THE MATRIX TO BE SEARCHED
                      ROWS - NUMBER OF ROWS IN THE MATRIX
                      COLS - NUMBER OF COLUMNS IN THE MATRIX
          */
 2     1  FIND$MX: PROCEDURE (MATRIX$PTR, ROWS, COLS) INTEGER EXTERNAL;
 3     2  DECLARE (ROWS, COLS) INTEGER;
 4     2  DECLARE MATRIX$PTR POINTER;
 5     2  END FIND$MX;


          /*  BIN$DEC$ASC - BINARY TO DECIMAL ASCII CONVERSION PROCEDURE
                  PARAMETERS:
                      VALUE - INTEGER VALUE TO BE CONVERTED TO ASCII
                      CHAR$ARRAY$ADR - ADDRESS OF 6 BYTE ARRAY WHERE ASCII
                          STRING CONTAINING THE VALUE WILL BE STORED
          */
 6     1  BIN$DEC$ASC: PROCEDURE (VALUE, CHAR$ARRAY$ADR);

 7     2  DECLARE (VALUE, TEMP, I) INTEGER;
 8     2  DECLARE CHAR$ARRAY$ADR POINTER;
 9     2  DECLARE (CHAR$ARRAY BASED CHAR$ARRAY$ADR) (6) BYTE;

10     2  IF VALUE < 0 THEN
11     2  DO;
12     3    CHAR$ARRAY(0) = '-';   /* SIGN CHARACTER */
13     3    TEMP = -VALUE;
14     3  END;
          ELSE
15     2  DO;
16     3    CHAR$ARRAY(0) = '+';
17     3    TEMP = VALUE;
18     3  END;
19     2  DO I = 5 TO 1 BY -1;
20     3    CHAR$ARRAY(I) = UNSIGN(TEMP MOD 10) + 30H;
21     3    TEMP = TEMP/10;
          /* ASCII CHARACTERS 30 THRU 39 HEX REPRESENT THE DIGITS 0 THRU 9. THUS
             TO CONVERT AN INTEGER TO ASCII REPEATED DIVISIONS BY 10 AND ADDING
             THE REMAINDER TO 30 HEX WILL ACCOMPLISH THE CONVERSION */
22     3  END;

23     2  END BIN$DEC$ASC;


          /*  CO - EXTERNAL PROCEDURE TO OUTPUT A CHARACTER TO THE SYSTEM CONSOLE.
                  THIS PROCEDURE IS PART OF THE ISBC 957 LIBRARY FOR CONSOLE I/O
                  PARAMETER:
                      CHAR - ASCII CHARACTER TO BE OUTPUT ON THE CONSOLE
          */
24     1  CO: PROCEDURE (CHAR) EXTERNAL;
25     2  DECLARE CHAR BYTE;
26     2  END CO;


             /* MATRIX DIMENSIONS */
27     1  DECLARE M LITERALLY '5';
28     1  DECLARE N LITERALLY '5';
29     1  DECLARE P LITERALLY '3';

          /* THE THREE MATRICES ARE DECLARED AS ARRAYS OF STRUCTURES.  X$ROW IS COMPOSED
             OF M STRUCTURES EACH OF WHICH IS COMPOSED OF N INTEGER ELEMENTS.  THUS
             X$ROW MAY BE THOUGHT OF AS A M X N MATRIX.  THE MATRIX WILL BE STORED AS
             A ROW-ORDER MATRIX WITH THE ELEMENTS OF EACH ROW STORED IN ADJACENT MEMORY
             LOCATIONS.  Y$ROW IS DECLARED AS A N X P MATRIX AND Z$ROW AS A N X P MATRIX */
30     1  DECLARE X$ROW(M) STRUCTURE (COL(N) INTEGER);
31     1  DECLARE Y$ROW(N) STRUCTURE (COL(P) INTEGER);
32     1  DECLARE Z$ROW(M) STRUCTURE (COL(P) INTEGER);
33     1  DECLARE (I,J,K,MAX) INTEGER;
34     1  DECLARE MAX$ASC$ARRAY(6) BYTE;
35     1  DECLARE TEXT(*) BYTE DATA ('MAX VALUE = ');
```

```
                    /* INITIALIZE X$ROW SUCH THAT THE FIRST ROW IS SET EQUAL TO 0, THE SECOND
                       ROW EQUAL TO 1, THE THIRD ROW EQUAL TO 2, ETC.  */
     36   1          DO I = 0 TO (M-1);
     37   2            DO J = 0 TO (N-1);
     38   3              X$ROW(I).COL(J) = I;
     39   3            END;
     40   2          END;


                    /* INITIALIZE Y$ROW SUCH THAT THE FIRST COLUMN IS SET EQUAL TO 0, THE
                       SECOND COLUMN EQUAL TO -1, AND THE THIRD COLUMN EQUAL TO -2.  */
     41   1          DO I = 0 TO (N-1);
     42   2            DO J = 0 TO (P-1);
     43   3              Y$ROW(I).COL(J) = -J;
     44   3            END;
     45   2          END;
                    /* PERFORM MATRIX MULTIPLICATION  */
     46   1          DO K = 0 TO (P-1);
     47   2            DO I = 0 TO (M-1);
     48   3              Z$ROW(I).COL(K) = 0;  /* SET Z$ROW ELEMENT TO 0 */
     49   3              DO J = 0 TO (N-1); /* SUM THE PRODUCT OF X$ROW ROW TERMS AND Y$ROW COLUMN TERMS */
     50   4                Z$ROW(I).COL(K) = Z$ROW(I).COL(K) + ( X$ROW(I).COL(J) * Y$ROW(J).COL(K) );
     51   4              END;
     52   3            END;
     53   2          END;

     54   1          MAX = FIND$MX (@Z$ROW, M, P); /* FIND MAX VALUE OF Z$ROW */

     55   1          CALL BIN$DEC$ASC (MAX, @MAX$ASC$ARRAY); /* CONVERT TO DECIMAL ASCII */

     56   1          DO I = 0 TO (SIGNED(SIZE(TEXT)) - 1); /* OUTPUT HEADER TEXT */
     57   2            CALL CO(TEXT(I));
     58   2          END;

     59   1          DO I = 0 TO 5; /* OUTPUT ASCII MAX VALUE */
     60   2            CALL CO(MAX$ASC$ARRAY(I));
     61   2          END;


     62   1          END EXECUTION$VEHICLE;



        MODULE INFORMATION:

            CODE AREA SIZE     = 0225H     549D
            CONSTANT AREA SIZE = 000CH      12D
            VARIABLE AREA SIZE = 0090H     144D
            MAXIMUM STACK SIZE = 0008H       8D
            137 LINES READ
            0 PROGRAM ERROR(S)

        END OF PL/M-86 COMPILATION

        ISIS-II MCS-86 ASSEMBLER      ASSEMBLY OF MODULE FIND
        OBJECT MODULE PLACED IN :F1:FIND.OBJ
        ASSEMBLER INVOKED BY: ASM86 :F1:FIND.ASM DEBUG

        LOC  OBJ


           LINE    SOURCE

             1              NAME    FIND
             2              PUBLIC  FINDMX
             3      ;
             4      ;
             5      ;
             6      ;                          FINDMX
             7      ;      ASSEMBLY LANGUAGE PROCEDURE TO FIND THE ELEMENT OF AN INTEGER
             8      ;      MATRIX WITH THE LARGEST ABSOLUTE MAGNITUDE.  THE VALUE OF THE
             9      ;      ELEMENT IS RETURNED IN THE AX REGISTER.
            10      ;
            11      ;      PL/M CALLING SEQUENCE:
            12      ;              MAX$VALUE = FIND$MX(ADR$OF$MATRIX, #$OF$ROWS, #$OF$COLS);
            13      ;
            14      ;      PARAMETERS:
            15      ;              ADR$OF$MATRIX - ADDRESS OF THE MATRIX WHICH WILL BE SEARCHED
            16      ;              #$OF$ROWS - NUMBER OF ROWS IN THE MATRIX
            17      ;              #$OF$COLS - NUMBER OF COLUMNS IN THE MATRIX
            18      ;
            19      ;      PL/M WILL PASS THE THREE PARAMETERS IN THE CALL TO THIS PROCEDURE ON
            20      ;      THE STACK.  ON ENTRY TO THE PROCEDURE SP+6 WILL POINT TO THE FIRST
            21      ;      PARAMETER(ADR$OF$MATRIX) AND SP+4 AND SP+2 WILL POINT TO THE SECOND
            22      ;      AND THIRD PARAMETERS.
            23      ;
            24      ;      THE PROCEDURE IS A TYPED PROCEDURE WHICH ASSIGNS THE MAXIMUM VALUE
            25      ;      IN THE MATRIX TO A VARIABLE (IN THIS CASE MAX$VALUE) IN A PL/M
            26      ;      ASSIGNMENT STATEMENT.  TO ACCOMPLISH THIS ASSIGNMENT THE VALUE IS
            27      ;      RETURNED IN THE AX REGISTER.
            28      ;
            29      ;
            30      ;      THE ALGORITHM USED IS SIMILAR TO THE FOLLOWING PL/M CODE:
            31      ;              FOR I = 0 TO (#$OF$ROWS - 1);
            32      ;                FOR J = 0 TO (#$OF$COLS - 1);
            33      ;                  IF IABS(MATRIX(I).Y(J)) > IABS(MAX) THEN MAX = MATRIX(I).Y(J);
            34      ;                END;
            35      ;              END;
            36      ;
            37      ;              WHERE IABS(XYZ) REPRESENTS THE ABSOLUTE VALUE OF THE INTEGER XYZ
            38      ;
            39      ;
```

Ⓑ Ⓒ Ⓓ Ⓔ Ⓕ Ⓗ

```
       LOC  OBJ              LINE   SOURCE

                              40   ;
                              41   ;        DEFINE GROUPS TO CONFORM WITH PL/M-86 CONVENTIONS. DATA, STACK, AND
                              42   ;        CODE SEGMENTS WILL BE APPENDED TO THEIR RESPECTIVE SEGMENTS IN THE
                              43   ;        PL/M-86 MODULES.
                              44   DGROUP   GROUP    DATA,STACK
                              45   CGROUP   GROUP    CODE
                              46   ;
                              47   ;        INSTRUCT THE ASSEMBLER THAT THE DS, SS, AND CS REGISTERS WILL CONTAIN
                              48   ;        THE BASE ADDRESS VALUES FOR THE DGROUP, DGROUP AND CGROUP GROUPS.
                              49            ASSUME   DS:DGROUP,SS:DGROUP,CS:CGROUP
                              50   ;
                              51   ;
                              52   ;
                              53   ;***************DATA SEGMENT
                              54   ;
       ----                   55   DATA     SEGMENT WORD PUBLIC 'DATA'
       0000 0000              56   MAX      DW       0
       ----                   57   DATA     ENDS
                              58   ;
                              59   ;***************STACK SEGMENT
                              60   ;
       ----                   61   STACK    SEGMENT STACK 'STACK'
       0000 (14               62            DW       14 DUP (0)       ;RESERVE 13 WORDS OF STACK FOR MONITOR
            0000
            )
                              63                                     ;AND 1 WORD FOR FINDMX PROCEDURE
       ----                   64   STACK    ENDS
                              65   ;
                              66   ;***************CODE SEGMENT
                              67   ;
       ----                   68   CODE     SEGMENT BYTE PUBLIC 'CODE'
                              69   ;
                              70   ;PARAMETERS ON STACK, DISPLACEMENT FROM TOS INCREASED BY TWO DUE TO INITIAL PUSH
       0006[]                 71   NO_OF_ROWS    EQU       WORD PTR [BP+6]
       0004[]                 72   NO_OF_COLS    EQU       WORD PTR [BP+4]
       0008[]                 73   ADR_OF_MATRIX EQU       WORD PTR [BP+8]
                              74   ;
       0000                   75   FINDMX   PROC     NEAR            ;PROCEDURE DECLARATION
       0000 55                76            PUSH     BP              ;SAVE BP REGISTER
       0001 8BEC              77            MOV      BP,SP           ;BP POINTS TO PARAMETERS ON STACK
       0003 33D2              78            XOR      DX,DX           ;SET DX = ABS OF CURRENT MAX = 0
       0005 8BFA              79            MOV      DI,DX           ;DI = I (ROW INDEX) = 0
       0007 8BF2              80            MOV      SI,DX           ;SI = J (COLUMN INDEX) = 0
       0009 89160000     R    81            MOV      MAX,DX          ;MAX = CURRENT MAX = 0
       000D 8B4E04            82            MOV      CX,NO_OF_COLS   ;CX = (#$OF$COLS) * 2
       0010 D1E1              83            SHL      CX,1
                              84                                    ;TERMINATION FOR J(SI) INDEX
       0012 8B5E08            85            MOV      BX,ADR_OF_MATRIX ;ADR$OF$MATRIX PARAMETER
                              86                                    ;BX POINTS TO FIRST ELEMENT OF A GIVEN ROW
       0015 8B00              87   ABC:     MOV      AX,[BX][SI]     ;GET ELEMENT OF MATRIX
       0017 0BC0              88            OR       AX,AX           ;SET FLAGS
       0019 7902              89            JNS      DEF             ;JUMP IF SIGN = 0
       001B F7D8              90            NEG      AX              ;NEGATE TO FORM POSITIVE NUMBER
       001D 3BC2              91   DEF:     CMP      AX,DX           ;COMPARE TO CURRENT MAX
       001F 7C07              92            JL       XYZ             ;JUMP IF LESS THAN CURRENT MAX
       0021 8BD0              93            MOV      DX,AX           ;MOVE TO ABS OF CURRENT MAX
       0023 8B00              94            MOV      AX,[BX][SI]     ;MOVE MATRIX VALUE TO CURRENT MAX
       0025 A30000       R    95            MOV      MAX,AX
       0028 83C602            96   XYZ:     ADD      SI,2            ;INCREMENT J INDEX BY TWO
       002B 3BF1              97            CMP      SI,CX           ;END OF THIS ROW ??
       002D 72E6              98            JB       ABC             ;IF NO, LOOP BACK FOR NEXT ELEMENT OF THIS ROW
       002F 8D18              99            LEA      BX,[BX+SI]      ;BX = BX + (2 * #$OF$COLS), BX POINTS TO NEXT ROW
       0031 BE0000           100            MOV      SI,0            ;J = 0
       0034 47               101            INC      DI              ;I = I + 1
       0035 3B7E06           102            CMP      DI,NO_OF_ROWS   ;LAST ROW ??
       0038 72DB             103            JB       ABC             ;IF NO, DO THE NEXT ROW
       003A A10000       R   104            MOV      AX,MAX          ;RETURN MAX VALUE IN AX REGISTER
       003D 5D               105            POP      BP              ;RESTORE BP REGISTER
       003E C20600          106            RET      6               ;INCREMENT SP BY 6 AND RETURN TO CALLER
                            107   ;
                            108   FINDMX   ENDP
                            109   ;
       ----                 110   CODE     ENDS
                            111   ;
                            112            END
```

SYMBOL TABLE LISTING
------ ----- -------

NAME	TYPE	VALUE	ATTRIBUTES
??SEG	SEGMENT		SIZE=0000H PARA PUBLIC
ABC	L NEAR	0015H	CODE
ADR_OF_MATRIX	V WORD	0008H	[BP]
CGROUP. . . .	GROUP		CODE
CODE.	SEGMENT		SIZE=0041H BYTE PUBLIC 'CODE'
DATA.	SEGMENT		SIZE=0002H WORD PUBLIC 'DATA'
DEF	L NEAR	001DH	CODE
DGROUP. . . .	GROUP		DATA STACK
FINDMX. . . .	L NEAR	0000H	CODE PUBLIC
MAX	V WORD	0000H	DATA
NO_OF_COLS. .	V WORD	0004H	[BP]
NO_OF_ROWS. .	V WORD	0006H	[BP]
STACK	SEGMENT		SIZE=001CH PARA STACK 'STACK'
XYZ	L NEAR	0028H	CODE

ASSEMBLY COMPLETE, NO ERRORS FOUND

```
ISIS-II QRL-86, V1.1

INVOKED BY:
QRL86 :F1:MATRIX.OBJ,:F1:FIND.OBJ,SBCIOS.LIB ORIGIN(100CH)

INPUT MODULES INCLUDED:
:F1:MATRIX.OBJ(EXECUTIONVEHICLE)
:F1:FIND.OBJ(FIND)
SBCIOS.LIB(SBCCO)

RESULT WRITTEN TO :F1:MATRIX(EXECUTIONVEHICLE)
START ADDRESS IS (0100H,0002H)

 START   LTH  ALIGN NAME                        CLASS

 0100H   2A0H   G   /GS/ CGROUP
 0100H   225H   W   CODE(EXECUTIONVEHICLE)      CODE
 01225H   41H   B   CODE(FIND)                  CODE
 0126GH   3AH   W   CODE(SBCCO)                 CODE
                    /GE/ CGROUP
 012A0H   D0H   G   /GS/ DGROUP
 012A0H   CH    W   CONST(EXECUTIONVEHICLE)     CONST
 012ACH   0H    W   CONST(SBCCO)                CONST
 012ACH   90H   W   DATA(EXECUTIONVEHICLE)      DATA
 0133CH   2H    W   DATA(FIND)                  DATA
 0133EH   0H    W   DATA(SBCCO)                 DATA
 01340H   30H   SW  STACK                       STACK
 01370H   0H    W   MEMORY                      MEMORY
                    /GE/ DGROUP
 0 370H   0H    G   ??SEG(FIND)                 (NULL)
```

```
DEBUG MAP OF :F1:MATRIX(EXECUTIONVEHICLE)

              MODULE: EXECUTIONVEHICLE  0100H,01E1H  LINE #: 19        0100H,0139H  LINE #: 52
012AH,00D0H  SYMBOL: MEMORY            0100H,01F8H  LINE #: 20        0100H,0142H  LINE #: 53
0100H,01B5H  SYMBOL: BINDECASC         0100H,0213H  LINE #: 21        0100H,014BH  LINE #: 54
012AH,000CH  SYMBOL: TEMP              0100H,021EH  LINE #: 22        0100H,015EH  LINE #: 55
012AH,000EH  SYMBOL: I                 0100H,0221H  LINE #: 23        0100H,0169H  LINE #: 56
012AH,0010H  SYMBOL: XROW              0100H,0002H  LINE #: 36        0100H,017AH  LINE #: 57
012AH,004CH  SYMBOL: YROW              0100H,0021H  LINE #: 37        0100H,0185H  LINE #: 58
012AH,006AH  SYMBOL: ZROW              0100H,0032H  LINE #: 38        0100H,018EH  LINE #: 59
012AH,008EH  SYMBOL: I                 0100H,004BH  LINE #: 39        0100H,019FH  LINE #: 60
012AH,0090H  SYMBOL: J                 0100H,0054H  LINE #: 40        0100H,01AAH  LINE #: 61
012AH,0092H  SYMBOL: K                 0100H,005DH  LINE #: 41        0100H,01B3H  LINE #: 62
012AH,0094H  SYMBOL: MAX               0100H,006EH  LINE #: 42                     MODULE: FIN
012AH,0096H  SYMBOL: MAXASCARRAY       0100H,007FH  LINE #: 43        0100H,023AH  SYMBOL: ABC
012AH,0000H  SYMBOL: TEXT              0100H,009CH  LINE #: 44        0100H,0242H  SYMBOL: DEF
0100H,01B5H  LINE #: 6                 0100H,00A5H  LINE #: 45        0100H,0225H  SYMBOL: FINDMX
0100H,01B8H  LINE #: 10                0100H,00AEH  LINE #: 46        012AH,009CH  SYMBOL: MAX
0100H,01C2H  LINE #: 12                0100H,00BFH  LINE #: 47        0100H,024DH  SYMBOL: XYZ
0100H,01C8H  LINE #: 13                0100H,00D0H  LINE #: 48        0100H,0225H  PUBLIC: FINDMX
0100H,01D1H  LINE #: 14                0100H,00E7H  LINE #: 49                     MODULE: SBCCO
0100H,01D4H  LINE #: 16                0100H,00F8H  LINE #: 50        0100H,0266H  PUBLIC: CO
0100H,01DAH  LINE #: 17                0100H,0130H  LINE #: 51
```

APPENDIX C

PROGRAM LISTING FOR EXECUTION$VEHICLE MODULE WITH CODE EXPANSION

```
PL/M-86 COMPILER    EXECUTIONVEHICLE


ISIS-II PL/M-86 V1.0 COMPILATION OF MODULE EXECUTIONVEHICLE
NO OBJECT MODULE REQUESTED
COMPILER INVOKED BY:   PLM86 :F1:MATRIX.PLM DEBUG CODE NOOBJECT PRINT(:F1:MATRIX.XLS)



                 /*      MATRIX MULTIPLICATION EXAMPLE PROGRAM

                     PL/M-86 MAIN PROGRAM WHICH:
                         A) INITIALIZES TWO INTEGER MATRICES
                         B) MULTIPLIES THE TWO MATRICES AND STORES THE RESULT IN A
                            THIRD MATRIX
                         C) CALLS AN ASSEMBLY LANGUAGE PROCEDURE WHICH SEARCHES THE
                            THIRD MATRIX FOR THE MAXIMUM VALUE
                         D) CALLS A PL/M PROCEDURE WHICH CONVERTS THE MAXIMUM VALUE
                            FROM INTEGER TO ASCII
                         E) CALLS A PROCEDURE WHICH OUTPUTS THE ASCII CHARACTERS ON
                            THE SYSTEM CONSOLE
                 */

   1             EXECUTION$VEHICLE:
                 DO;


                 /*  FIND$MX - EXTERNAL ASSEMBLY LANGUAGE PROCEDURE WHICH SEARCHES A
                            MATRIX FOR THE LARGEST ABSOLUTE MAGNITUDE.
                         PARAMETERS:
                             MATRIX$ADR -  ADDRESS OF THE MATRIX TO BE SEARCHED
                             ROWS - NUMBER OF ROWS IN THE MATRIX
                             COLS - NUMBER OF COLUMNS IN THE MATRIX
                 */
   2     1       FIND$MX: PROCEDURE (MATRIX$PTR, ROWS, COLS) INTEGER EXTERNAL;
   3     2       DECLARE (ROWS, COLS) INTEGER;
   4     2       DECLARE MATRIX$PTR POINTER;
   5     2       END FIND$MX;



                 /*  BIN$DEC$ASC - BINARY TO DECIMAL ASCII CONVERSION PROCEDURE
                         PARAMETERS:
                             VALUE - INTEGER VALUE TO BE CONVERTED TO ASCII
                             CHAR$ARRAY$ADR - ADDRESS OF 6 BYTE ARRAY WHERE ASCII
                                   STRING CONTAINING THE VALUE WILL BE STORED
                 */
   6     1       BIN$DEC$ASC: PROCEDURE (VALUE, CHAR$ARRAY$ADR);
                                           ; STATEMENT # 6
                              BINDECASC       PROC NEAR
             01B5  55                  PUSH    BP
             01B6  8BEC                MOV     BP,SP

   7     2       DECLARE (VALUE, TEMP, I) INTEGER;
   8     2       DECLARE CHAR$ARRAY$ADR POINTER;
   9     2       DECLARE (CHAR$ARRAY BASED CHAR$ARRAY$ADR) (6) BYTE;

  10     2       IF VALUE < 0 THEN
                                           ; STATEMENT # 10
             01B8  817E060000          CMP     [BP].VALUE,0H

             01BD  7C03                JL      $+5H
             01BF  E91200              JMP     @1
  11     2       DO;
  12     3          CHAR$ARRAY(0) = '-';   /* SIGN CHARACTER */
                                           ; STATEMENT # 12
             01C2  8B5E04              MOV     BX,[BP].CHARARRAYADR
             01C5  C6072D              MOV     CHARARRAY[BX],2DH
  13     3          TEMP = -VALUE;
                                           ; STATEMENT # 13
             01C8  8B4606              MOV     AX,[BP].VALUE
             01CB  F7D8                NEG     AX
             01CD  890600C0            MOV     TEMP,AX
  14     3          END;
                                           ; STATEMENT # 14
             01D1  E90D00              JMP     @2
                              @1:
                 ELSE
  15     2       DO;
  16     3          CHAR$ARRAY(0) = '+';
                                           ; STATEMENT # 16
             01D4  8B5E04              MOV     BX,[BP].CHARARRAYADR
             01D7  C6072B              MOV     CHARARRAY[BX],2BH
  17     3          TEMP = VALUE;
                                           ; STATEMENT # 17
             01DA  8B4606              MOV     AX,[BP].VALUE
             01DD  890600C0            MOV     TEMP,AX
  18     3          END;
                              @2:
  19     2       DO I = 5 TO 1 BY -1;
                                           ; STATEMENT # 19
             01E1  C7060200050C        MOV     I,5H
             01E7  E90600              JMP     @5
                              @3:
             01EA  810602F0FFFF        ADD     I,0FFFFH
```

```
                        @5:
            01F0  813E02000100    CMP     I,1H
            01F6  7D03            JGE     $+5H
            01F8  E92600          JMP     @4
20   3          CHARSARRAY(I) = UNSIGN(TEMP MOD 10) + 30H;
                                            ; STATEMENT # 20
            01FB  8B060000        MOV     AX,TEMP
            01FF  B90A00          MOV     CX,0AH
            0202  31D2            XOR     DX,DX
            0204  F7F9            IDIV    CX
            0206  81C23000        ADD     DX,30H
            020A  8B5E04          MOV     BX,[BP].CHARARRAYADR
            020D  8B3602000       MOV     SI,I
            0211  8810            MOV     [BX].CHARARRAY[SI],DL
21   3          TEMP = TEMP/10;
                                            ; STATEMENT # 21
                /* ASCII CHARACTERS 30 THRU 39 HEX REPRESENT THE DIGITS 0 THRU 9. THUS
                   TO CONVERT AN INTEGER TO ASCII REPEATED DIVISIONS BY 10 AND ADDING
                   THE REMAINDER TO 30 HEX WILL ACCOMPLISH THE CONVERSION */
            0213  8B060000        MOV     AX,TEMP
            0217  99              CWD
            0218  F7F9            IDIV    CX
            021A  89060000        MOV     TEMP,AX
22   3      END;
                                            ; STATEMENT # 22
            021E  E9C9FF          JMP     @3
                        @4:

23   2      END BINSDECSASC;
                                            ; STATEMENT # 23
            0221  5D              POP     BP
            0222  C20400          RET     4H
                        BINDECASC   ENDP


                /*  CO - EXTERNAL PROCEDURE TO OUTPUT A CHARACTER TO THE SYSTEM CONSOLE.
                       THIS PROCEDURE IS PART OF THE ISBC 957 LIBRARY FOR CONSOLE I/O
                       PARAMETER:
                           CHAR - ASCII CHARACTER TO BE OUTPUT ON THE CONSOLE
                    */
24   1      CO: PROCEDURE (CHAR) EXTERNAL;
25   2      DECLARE CHAR BYTE;
26   2      END CO;


                /* MATRIX DIMENSIONS */
27   1      DECLARE M LITERALLY '6';
28   1      DECLARE N LITERALLY '5';
29   1      DECLARE P LITERALLY '3';

                /* THE THREE MATRICES ARE DECLARED AS ARRAYS OF STRUCTURES. XSROW IS COMPOSED
                   OF M STRUCTURES EACH OF WHICH IS COMPOSED OF N INTEGER ELEMENTS.  THUS
                   XSROW MAY BE THOUGHT OF AS A M X N MATRIX. THE MATRIX WILL BE STORED AS
                   A ROW-ORDER MATRIX WITH THE ELEMENTS OF EACH ROW STORED IN ADJACENT MEMORY
                   LOCATIONS.  YSROW IS DECLARED AS A N X P MATRIX AND ZSROW AS A N X P MATRIX */
30   1      DECLARE XSROW(M) STRUCTURE (COL(N) INTEGER);
31   1      DECLARE YSROW(N) STRUCTURE (COL(P) INTEGER);
32   1      DECLARE ZSROW(N) STRUCTURE (COL(P) INTEGER);

33   1      DECLARE (I,J,K,MAX) INTEGER;
34   1      DECLARE MAXSASCSARRAY(6) BYTE;
35   1      DECLARE TEXT(*) BYTE DATA ('MAX VALUE = ');


                /* INITIALIZE XSROW SUCH THAT THE FIRST ROW IS SET EQUAL TO 0, THE SECOND
                   ROW EQUAL TO 1, THE THIRD ROW EQUAL TO 2, ETC.  */
36   1      DO I = 0 TO (M-1);
                                            ; STATEMENT # 36
            0002  FA              CLI
            0003  2E8E160000      MOV     SS,CS:@@STACKSFRAME
            0008  BC0800          MOV     SP,@@STACKSOFFSET
            000B  8BEC            MOV     BP,SP
            000D  16              PUSH    SS
            000E  1F              POP     DS
            000F  FB              STI
            0010  C706820000      MOV     I,0H
                        @6:
            0016  813E8200000500  CMP     I,5H
            001C  7E03            JLE     $+5H
            001E  E93C00          JMP     @7
37   2          DO J = 0 TO (N-1);
                                            ; STATEMENT # 37
            0021  C70684000000    MOV     J,0H
                        @8:
            0027  813E84000400    CMP     J,4H
            002D  7E03            JLE     $+5H
            002F  E92200          JMP     @9
38   3              XSROW(I).COL(J) = I;
                                            ; STATEMENT # 38
            0032  8B068200        MOV     AX,I
            0036  B90A00          MOV     CX,0AH
            0039  F7E9            IMUL    CX
            003B  8B368400        MOV     SI,J
            003F  D1E6            SHL     SI,1
            0041  89C3            MOV     BX,AX
            0043  8B0E8200        MOV     CX,I
            0047  89880400        MOV     [BX].XROW[SI],CX
39   3          END;
```

```
                                                         ; STATEMENT # 39
         004B 8106840001CC    ADD     J,1H
         0051 E9D3FF          JMP     @8
                      @9:
40   2     END;
                                                         ; STATEMENT # 40
         0054 8106820001C0    ADD     I,1H
         005A E9B9FF          JMP     @6
                      @7:


           /* INITIALIZE Y$ROW SUCH THAT THE FIRST COLUMN IS SET EQUAL TO 0, THE
              SECOND COLUMN EQUAL TO -1, AND THE THIRD COLUMN EQUAL TO -2.  */
41   1     DO I = 0 TO (N-1);
                                                         ; STATEMENT # 41
         005D C706820000CC    MOV     I,0H
                      @10:
         0063 813E82000400    CMP     I,4H
         0069 7E03            JLE     $+5H
         006B E94000          JMP     @11
42   2       DO J = 0 TO (P-1);
                                                         ; STATEMENT # 42
         006E C70684000000    MOV     J,0H
                      @12:
         0074 813E84000200    CMP     J,2H
         007A 7E03            JLE     $+5H
         007C E92600          JMP     @13
43   3         Y$ROW(I).COL(J) = -J;
                                                         ; STATEMENT # 43
         007F 8B068400        MOV     AX,J
         0083 F7D8            NEG     AX
         0085 50              PUSH    AX       ; 1
         0086 8B068200        MOV     AX,I
         008A B90600          MOV     CX,6H
         008D F7E9            IMUL    CX
         008F 8B368400        MOV     SI,J
         0093 D1E6            SHL     SI,1
         0095 89C3            MOV     BX,AX
         0097 59              POP     CX       ; 1
         0098 89884000        MOV     [BX].YROW[SI],CX
44   3         END;
                                                         ; STATEMENT # 44
         009C 8106840001C0    ADD     J,1H
         00A2 E9CFFF          JMP     @12
                      @13:
45   2       END;
                                                         ; STATEMENT # 45
         00A5 8106820001C0    ADD     I,1H
         00AB E9B5FF          JMP     @10
                      @11:
           /* PERFORM MATRIX MULTIPLICATION  */
46   1     DO K = 0 TO (P-1);
                                                         ; STATEMENT # 46
         00AE C70686000000    MOV     K,0H
                      @14:
         00B4 813E86000200    CMP     K,2H
         00BA 7E03            JLE     $+5H
         00BC E9CC00          JMP     @15
47   2       DO I = 0 TO (M-1);
                                                         ; STATEMENT # 47
         00BF C706820000CC    MOV     I,0H
                      @16:
         00C5 813E82000500    CMP     I,5H
         00CB 7E03            JLE     $+5H
         00CD E97200          JMP     @17
48   3         Z$ROW(I).COL(K) = 0;  /* SET Z$ROW ELEMENT TO 0 */
                                                         ; STATEMENT # 48
         00D0 8B068200        MOV     AX,I
         00D4 B90600          MOV     CX,6H
         00D7 F7E9            IMUL    CX
         00D9 8B368600        MOV     SI,K
         00DD D1E6            SHL     SI,1
         00DF 89C3            MOV     BX,AX
         00E1 C78050000000    MOV     [BX].ZROW[SI],0H
49   3         DO J = 0 TO (N-1); /* SUM THE PRODUCT OF X$ROW ROW TERMS AND Y$ROW COLUMN TERMS */
                                                         ; STATEMENT # 49
         00E7 C70684000000    MOV     J,0H
                      @18:
         00ED 813E84000400    CMP     J,4H
         00F3 7E03            JLE     $+5H
         00F5 E94100          JMP     @19
50   4           Z$ROW(I).COL(K) = Z$ROW(I).COL(K) + ( X$ROW(I).COL(J) * Y$ROW(J).COL(K) );
                                                         ; STATEMENT # 50
         00F8 8B068200        MOV     AX,I
         00FC B90A00          MOV     CX,0AH
         00FF F7E9            IMUL    CX
         0101 8B368400        MOV     SI,J
         0105 D1E6            SHL     SI,1
         0107 50              PUSH    AX       ; 1
         0108 8B068400        MOV     AX,J
         010C B90600          MOV     CX,6H
         010F F7E9            IMUL    CX
         0111 8B3E8600        MOV     DI,K
         0115 D1E7            SHL     DI,1
         0117 89C3            MOV     BX,AX
         0119 8B814000        MOV     AX,[BX].YROW[DI]
         011D 5B              POP     BX       ; 1
         011E F7A80400        IMUL    [BX].XROW[SI]
         0122 50              PUSH    AX       ; 1
         0123 8B068200        MOV     AX,I
         0127 F7E9            IMUL    CX
         0129 89C3            MOV     BX,AX
```

```
                    012B  58              POP     AX      ; 1
                    012C  011815E00       ADD     [BX].ZROW[DI],AX
        51    4        END;
                                                  ; STATEMENT # 51
                    0130  810684000100    ADD     J,1H
                    0136  E9B4FF          JMP     @18
                                  @19:
        52    3      END;
                                                  ; STATEMENT # 52
                    0139  810682000100    ADD     I,1H
                    013F  E983FF          JMP     @16
                                  @17:
        53    2     END;
                                                  ; STATEMENT # 53
                    0142  810686000100    ADD     K,1H
                    0148  E969FF          JMP     @14
                                  @15:

        54    1     MAX = FINDSMX (@Z$ROW, M, P); /* FIND MAX VALUE OF Z$ROW */
                                                  ; STATEMENT # 54
                    014B  B85E00          MOV     AX,OFFSET(ZROW)
                    014E  50              PUSH    AX      ; 1
                    014F  B80600          MOV     AX,6H
                    0152  50              PUSH    AX      ; 2
                    0153  B80300          MOV     AX,3H
                    0156  50              PUSH    AX      ; 3
                    0157  E80000          CALL    FINDMX
                    015A  89068800        MOV     MAX,AX
        55    1     CALL BIN$DEC$ASC (MAX, @MAX$ASC$ARRAY); /* CONVERT TO DECIMAL ASCII */
                                                  ; STATEMENT # 55
                    015E  FF368800        PUSH    MAX     ; 1
                    0162  B88A00          MOV     AX,OFFSET(MAXASCARRAY)
                    0165  50              PUSH    AX      ; 2
                    0166  E84C00          CALL    BINDECASC

        56    1     DO I = 0 TO (SIGNED(SIZE(TEXT)) - 1); /* OUTPUT HEADER TEXT */
                                                  ; STATEMENT # 56
                    0169  C70682000000    MOV     I,0H
                                  @20:
                    016F  813E82000B00    CMP     I,0BH
                    0175  7E03            JLE     $+5H
                    0177  E91400          JMP     @21
        57    2       CALL CO(TEXT(I));
                                                  ; STATEMENT # 57
                    017A  8B1E8200        MOV     BX,I
                    017E  FFB70000        PUSH    TEXT[BX]; 1
                    0182  E80000          CALL    CO
        58    2     END;
                                                  ; STATEMENT # 58
                    0185  810682000100    ADD     I,1H
                    018B  E9E1FF          JMP     @20
                                  @21:

        59    1     DO I = 0 TO 5; /* OUTPUT ASCII MAX VALUE */
                                                  ; STATEMENT # 59
                    018E  C70682000000    MOV     I,0H
                                  @22:
                    0194  813E82000500    CMP     I,5H
                    019A  7E03            JLE     $+5H
                    019C  E91400          JMP     @23
        60    2       CALL CO(MAX$ASC$ARRAY(I));
                                                  ; STATEMENT # 60
                    019F  8B1E8200        MOV     BX,I
                    01A3  FFB78A00        PUSH    MAXASCARRAY[BX]; 1
                    01A7  E80000          CALL    CO
        61    2     END;
                                                  ; STATEMENT # 61
                    01AA  810682000100    ADD     I,1H
                    01B0  E9E1FF          JMP     @22
                                  @23:


        62    1     END EXECUTION$VEHICLE;
                                                  ; STATEMENT # 62
                    01B3  FB              STI
                    01B4  F4              HLT




MODULE INFORMATION:

    CODE AREA SIZE     = 0225H     549D
    CONSTANT AREA SIZE = 000CH      12D
    VARIABLE AREA SIZE = 0090H     144D
    MAXIMUM STACK SIZE = 0008H       8D
    137 LINES READ
    0 PROGRAM ERROR(S)

END OF PL/M-86 COMPILATION
```

intel

Using the iSBC 544™ Intelligent Communications Controller

Steve Verleye
OEM Microcomputer Systems Applications

Using the
iSBC 544 Intelligent
Communications Controller

Contents

I. INTRODUCTION.

As the microcomputer system found its way into more and more demanding applications the need became clear for a new and innovative solution to the old problem of providing timely response to real world events. This need was never clearer than in the field of communications where throughput and response time are the keys to success. The iSBC 544 Intelligent Communications Controller (ICC) is the vanguard of a family of intelligent slave computers that provide a unique and powerful answer to the needs of the microcomputer user.

This application note is intended to introduce the reader to the intelligent slave concept in general and the iSBC 544 board in particular. After a brief overview of the evolution of the concept and the features it provides, the hardware and software aspects of the controller are studied. Following this a summary of various system throughput tests is examined to evaluate the performance of the intelligent slave versus more traditional system architectures. We then study two example applications of the product and relate the earlier discussions to the real world. Finally, some system software is presented that handles all data transfer duties between master single board computers and intelligent slaves on the MULTIBUS system bus. More detailed information on many of the topics covered in this note can be found in the related publications listed in the front-piece.

II. OVERVIEW

Intelligent Slave Architecture

Over the years, component technology has increased at a rapid pace going from discrete components (eg. transistors) to integrated circuits (eg. TTL devices) to programmable peripheral controllers (eg. Intel 8251A Universal Synchronous/Asynchronous Receiver/Transmitter) to fully intelligent slave devices (eg. Intel 8041A Universal Peripheral Interface). At the system level the evolution followed a similar path using the increasing component technology to create more and more powerful system building blocks. The iSBC 508 I/O board used TTL logic to provide digital I/O expansion for iSBC computers. The

iSBC 534 board took advantage of programmable LSI devices to provide a programmable communications expansion board. Now, with the advent of the iSBC 544 Intelligent Communications Controller, a new level of system capability is made possible with the fully intelligent slave controller.

The cornerstone of the intelligent slave architecture is the dual port memory. Through the use of this shared memory space, a fast and efficient protocol can be established to allow for cooperation between master and intelligent slave in solving the needs of the application system. In addition to the shared memory, the CPU on the intelligent slave also has some local RAM and local PROM storage for programs. By using this architecture the advantages of multiprocessing and Direct Memory Access (DMA) controllers are blended together. Unlike DMA controllers, the intelligent slave works totally within its own data space. Therefore, it is not affected by bus traffic nor does it add to this traffic. And, since the onboard CPU gets its instructions from local PROM instead of predefined hard-wired logic or microcode, the user has total flexibility in defining the functions the intelligent slave will assume in the overall system.

Although the contents of an intelligent slave make it look very similar to a single board computer, the assumption of the slave role provides a distinct advantage. By performing duties for a master single board computer, the slave relieves the master of low-level processing duties and at the same time is itself relieved of system responsibilities.

In order to position the iSBC 544 product and outline what features it brings to the application system it is necessary to define the functions involved with communicating data. The three main functional divisions are illustrated in Figure 1. At the lowest level the physical interconnection is maintained. This level involves such standards as RS232C which defines the requirements for transmitting bits from point to point.

The data transmission level involves the transfer of bytes and/or blocks of data from devices to computers and from node to node in computer networks. The hardware dependent software such as interrupt service and device polling is
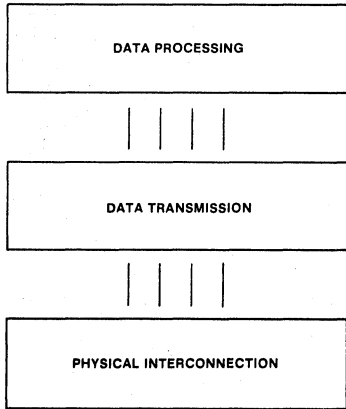
Figure 1. Layering of Communication System Functions

part of this level are as handlers for standard protocols such as SDLC, HDLC, Bisync and X.25 or special purpose schemes and custom protocols.

The highest level performs the actual processing of the data and calls upon the lower levels to move the data from place to place. The application software resides at this level as do some high level software functions such as program to program and process to process communications packages.

Now that we have a map of system functions to guide us, it is possible to gain an understanding of the usefulness of a product like the iSBC 544 Intelligent Communications Controller. If an iSBC 534 board (which contains four USART devices) was included to handle the expansion of serial I/O capacity the mapping of system functions would look like that shown in Figure 2. The four USARTs on the board would handle the physical interconnection but due to the lack of intelligence on the board the master CPU would be burdened with all of the data transmission duties in addition to its real duty, data processing.

When an iSBC 544 board is used in the system, the mapping of system functions is as shown in Figure 3. The physical interconnection is still handled by the USARTs on the board but now the on-board CPU can be programmed to assume the data transmission duties. With an intelligent slave in the system, the master CPU is freed to concentrate on the data processing functions and the end result is that each function in the system is handled in the most efficient manner possible.
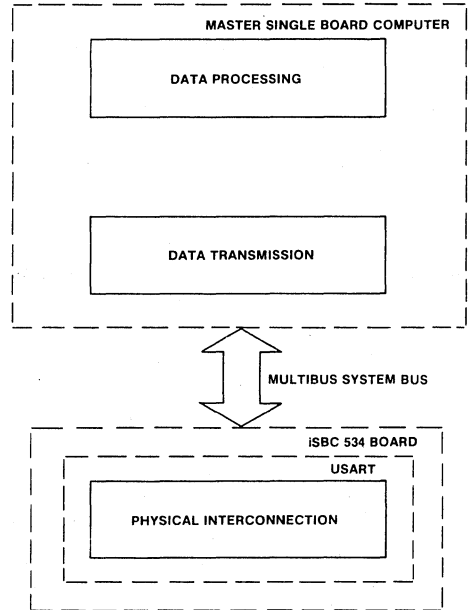

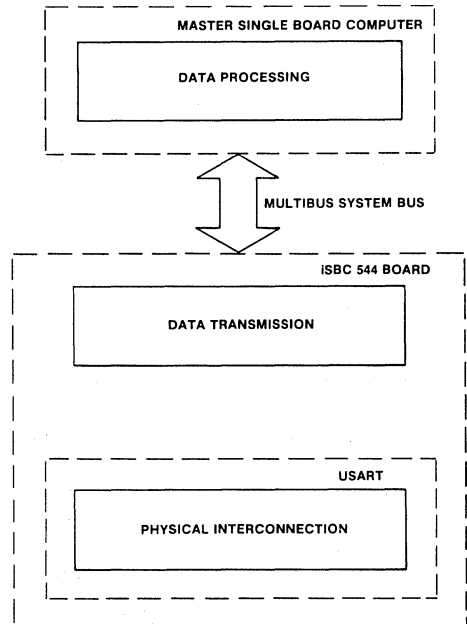
Figure 2. Mapping of System Functions with iSBC 534 Board

Figure 3. Mapping of System Functions with iSBC 544 Board

The iSBC 544 Board

The iSBC 544 Intelligent Communications Controller contains:

- An Intel 8085A CPU operating at 2.76 MHz.
- Sockets for up to 8K bytes of read only memory (user can choose Intel 2716, 2316E or 2732 devices).
- 16K bytes of dynamic, dual port Random Access Memory (RAM).
- 256 bytes of static local RAM.
- Four Intel 8251A USARTs with programmable baud rates.
- Two Intel 8253 Programmable Interval Timers.
- Intel 8155 parallel interface providing 22 parallel I/O lines and one 14 bit interval timer. Various input and output lines are dedicated to provide an interface to a Bell 801 or equivalent Automatic Call Unit (ACU).
- 8259A Priority Interrupt Controller.

III. HARDWARE CONSIDERATIONS

This section of the application note will focus on the iSBC 544 hardware and will outline the features of the board and its uses. Appendix A contains simplified logic diagrams of the iSBC 544 board which can be referenced in the following discussions.

Two Mode Operation

The iSBC 544 board is capable of operating in one of two modes; 1) intelligent slave and 2) stand-alone communications computer. The mode can either be set with a switch or it can be "toggled" via a software driven flip-flop on the board. In the intelligent slave mode the CPU on the iSBC 544 board operates strictly within its on-board resources. Communications with 8-bit and 16-bit master single board computers is accomplished through the dual port memory. Since the on-board CPU executes code out of its local PROM program storage the system designer is free to define which functions the slave will assume in the system design. As discussed earlier, this could include all or part of the system data transmission duties or could involve application specific duties such as terminal format control, code conversion or terminal input editing.

In the stand-alone mode, the logic on the board disables off board access to the dual port RAM and the bus buffers are used to allow the on-board CPU to access expansion memory and I/O on the MULTIBUS system bus. In this mode the iSBC 544 board drives the bus busy (BUSY/) control line active disallowing any other bus master access to the bus. The stand-alone communications computer is capable of performing all of the functions of the applications system. Referring once again to the diagram of the functions of a communication system, the stand-alone communications computer, with or without system expansion, is responsible for all data transmission and data processing functions. In small applications requiring multiple serial lines the stand-alone iSBC 544 controller is a perfect fit.

In very special circumstances it is possible to share the system bus by toggling the mode set flip-flop between master and slave mode. Figure 4
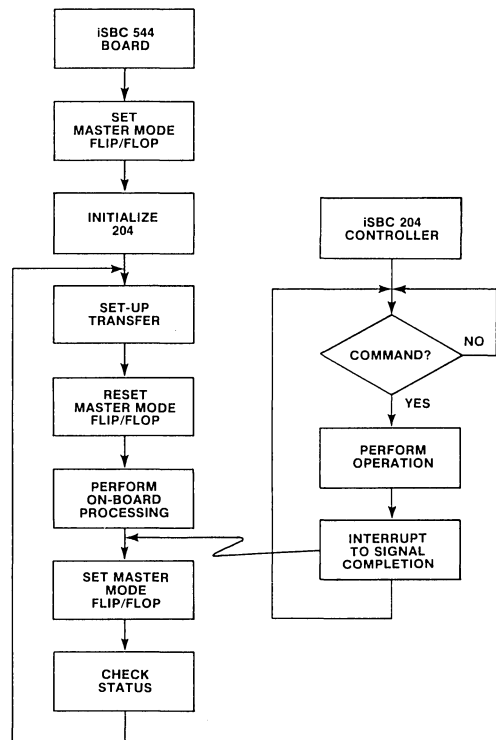

Figure 4. iSBC 544 Controller Running iSBC 204 Disk Controller

shows the flow chart for a routine (code in Appendix B) that makes use of the "software switch" to operate an iSBC 204 Diskette Controller. Using the iSBC 544 board in a system with DMA devices is not recommended except in cases where DMA accesses are short and relatively rare. The use of the CPU for the handling of other system devices could seriously degrade its performance as a communications controller. However, this capability could be extremely useful in a system such as a small message store and forward where the disk traffic is not heavy and including a CPU card just to handle the disk would be wasteful. Use of the "software switch" to share the bus with another iSBC CPU is not advised because of the amount of protocol that would be required to keep the CPUs from interfering with each other on the bus.

Dual Port RAM

Figure 5 illustrates the dual port RAM memory array on the iSBC 544 card. A triple bus architecture is used to allow other MULTIBUS bus masters access to the RAM on the intelligent slave. Both the on-board CPU's bus and the MULTIBUS system bus are connected to the dual

port controller. From here the dual port bus is connected to the 16K of dynamic RAM memory. Memory transfer requests from either of the first two busses are handled by the dual port control logic with the on-board CPU being given priority if contention arises. The local CPU is favored so that it is not overly delayed in handling its time critical functions.

The address mapping of the dual port memory on the iSBC 544 is diagrammed in Figure 6. The user can enable access from the MULTIBUS system bus to 0, 4K, 8K or all 16K of the RAM on each iSBC 544 board. The dual port control logic decodes the full 20-bit address and provides an 8-bit data path to the bus. For these reasons the iSBC 544 board is compatible with 8080A, 8085A and 8086 based single board computers. The user can also select the block of addresses on the system bus to which the iSBC 544 RAM will respond.

Figure 5. Dual Port Control Logic
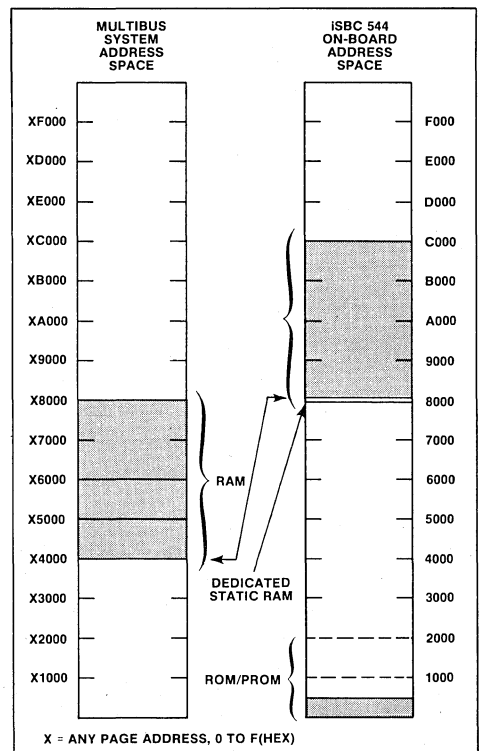
Figure 6. Address Mapping on Dual Port RAM Block

When accessed by the on-board CPU, the dual port RAM always appears at 8000H. If the iSBC 544 board is operating in the stand-alone computer mode, the board is capable of generating the 16-bit bus address supported by the 8085A CPU.

Interrupt Structure

The interrupt structure of the iSBC 544 controller is designed to handle the heavy load imposed by the inherent real-time nature of the communications application. An 8259A Priority Interrupt Controller handles the four receiver and transmitter ready interrupts from the 8251A devices and provides vectored interrupts using one of many available priority schemes. In addition to the eight interrupt sources handled by the 8259 there are various others that can be connected directly to the vectored interrupt inputs on the 8085A (RST 5.5, 6.5, 7.5 and TRAP). One interrupt is generated by the dual port control logic whenever a byte is written into the base address of the dual port memory by an offboard CPU. This interrupt, the flag interrupt, is cleared automatically when the on-board CPU reads the byte and is useful when designing a master-slave protocol since it provides a unique interrupt to each slave in the system.

If the 8251A devices are used to interface to modems the loss of carrier and ring indicator interrupts from all four channels need to be connected to 8085A interrupt request inputs. This is accomplished with four input OR gates tying the eight sources into RST 6.5. The ring indicator and carrier detect lines can also be monitored through a parallel I/O port. This port would be read in a polled system to determine status or could be used along with the OR-tied interrupts to determine which channel is sourcing the current interrupt.

The remaining interrupt sources come from the extra timer/counters and from the MULTIBUS interrupt lines. In addition to receiving interrupts from the bus, the iSBC 544 board has the capability of generating MULTIBUS interrupts using the Serial Output Data (SOD) line on the 8085A CPU.

Modem and Autocall Interface

The iSBC 544 controller uses 8251A and 8155 devices for interface to modems and an autocall unit respectively. All of the necessary handshaking signals concerned with the modem interface are connected to the 8251A and the carrier detect and ring indicator signals, as previously mentioned, can be connected to interrupt inputs. The 8155 parallel ports are wired as shown in Figure 7. All of the commonly used signals defined in the EIA RS-366 specification for interface to an autocall unit are provided. The software necessary for handling the ACU becomes a simple matter of responding to the ACU requests and sending out the BCD digits representing the number being dialed. In addition to the ACU interface, the 8155 monitors various signal states and provides software reset capabilities for the USARTs and some interrupts.

IV. SOFTWARE CONSIDERATIONS

Software for the iSBC 544 ICC falls into three main categories; device programming, master-slave protocols, and communications support. Each of these three topics is covered in the following section with the aim of defining the software requirements and functions of the iSBC 544 board.

Device Programming

The main sources of the power and flexibility of this product are the programmable LSI devices on the board. The first duty of the on-board software is programming these devices to handle the specific task at hand. To start with, the 8251A USART can be programmed for synchronous or asynchronous operation. In synchronous mode the user specifies even, odd or no parity and either external or internal sync detect with one or two sync characters. In the asynchronous mode the programmer selects the parity, the character length (5, 6, 7 or 8 data bits), the framing control (1, 1½ or 2 stop bits) and the baud rate scaling factor (input clock frequency divided by 1, 16 or 64).

The 8253 Programmable Interval Timers provide the receiver and transmitter clocks for the USARTs and, along with the 8251A baud rate scaling factor, are programmed by the software to provide the desired communications frequency. In addition, two additional 16 bit timers are left available to the applications programs to be used as event counters, real-time interrupts, etc.

PORT A

	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
OUTPUT PORT	CRQ	DPR	IR	UR	NB8	NB4	NB2	NB1

- NUMBER BIT NB1 (LSR): 1 = TRUE
- NUMBER BIT NB2: 1 = TRUE
- NUMBER BIT NB4: 1 = TRUE
- NUMBER BIT NB8 (MSB): 1 = TRUE
- USART RESET: 1 = TRUE
- INTERRUPT RESET: 1 = TRUE
- DIGIT PRESENT ON NUMBER BIT LINES: 0 = TRUE
- CALL REQUEST: 0 = TRUE

PORT B

	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
INPUT PORT	CD3	CD2	CD1	CD0	RI3	RI2	RI1	RI0

- RING INDICATOR, PORT 0: 0 = TRUE
- RING INDICATOR, PORT 1: 0 = TRUE
- RING INDICATOR, PORT 2: 0 = TRUE
- RING INDICATOR, PORT 3: 0 = TRUE
- CARRIER DETECT, PORT 0: 0 = TRUE
- CARRIER DETECT, PORT 1: 0 = TRUE
- CARRIER DETECT, PORT 2: 0 = TRUE
- CARRIER DETECT, PORT 3: 0 = TRUE

PORT C

	D_5	D_4	D_3	D_2	D_1	D_0
INPUT PORT	PFS	FINT	ACR	DLO	COS	PND

- PRESENT NEXT DIGIT: 0 = TRUE
- CALL COMPLETE, LINE TRANSFERRED TO MODEM: 0 = TRUE
- DATA LINE OCCUPIED: 0 = TRUE
- ABANDON CALL & RETRY: 0 = TRUE
- FLAG INTERRUPT: 1 = TRUE
- POWER FAIL SENSED: 1 = TRUE

Figure 7. 8155 Pinout Definitions

The 8259A Priority Interrupt Controller is programmed to vector all interrupts through a jump table in memory. Also, the device provides software selectable priority schemes and an interrupt mask register for sophisticated interrupt management designs.

Last, but not least, the 8155 Programmable Peripheral Interface provides various software controlled input and output ports as discussed in previous sections. One specific point to remember is that the power on state of the 8155 clamps the reset signal to the USARTs active and must be removed by programming the 8155 before communications can begin.

Master-Slave Protocols

If an application system is visualized at the highest level it appears to be a computer with various inputs and outputs as depicted in Figure 8a. If this computer is broken down into a master CPU and one or more intelligent slaves, great increases in efficiency and system throughput can be realized by distributing the duties between the CPUs (Figure 8b). Once this split is performed, some well defined means of communication between master and slaves needs to be defined so that the processes that execute on the different machines can cooperate. This means of communication takes the form of a protocol followed by both master and slave.

Figure 8a and 8b. System Software Block Diagrams

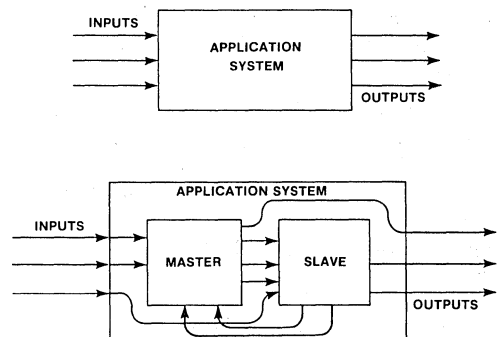The intelligent slave architecture was designed to simplify the development of the necessary protocol. The shared memory space in the dual port RAM provides a large communications buffer area where data and commands can be transferred using normal memory transfers. Data structures of any needed complexity can be built in this memory area and accessed by both master and slave. The flag interrupt can be used to provide a unique synchronization signal from a master to a given slave. In addition, the MULTI-BUS interrupt lines can be used to provide extra signals in both directions. As we shall see in the system software section, these basic tools can be utilized to design a general purpose data transfer mechanism which isolates the applications processes from the worries of protocols and synchronization.

Communications Support

The previous software topics dealt mainly with the system overhead that must be handled by the communications processor. The larger and more important duty of the CPU is dealing with the application at hand—communications.

When configured as an intelligent slave to some master iSBC CPU board, the iSBC 544 board works to offload the master of communications related functions and at the same time is itself relieved of a major share of the system overhead and can be tuned to provide the highest possible throughput. With this combination, more complex applications can be tackled where the number of lines and the line frequencies are greatly increased. Multiple systems can be employed to provide a network facility with the iSBC 544 board now handling the network protocol in addition to its other duties. The architecture of the iSBC 544 controller is designed to simplify the user's software development process. The board can be programmed to handle many possible data transmission functions from simple line protocols to terminal control to link protocols and all the way up to network protocols.

In the stand-alone mode, the iSBC 544 board can assume total responsibility for the application. This can be done with on-board resources only or can include the support of offboard expansion like the iSBC 534 four channel serial controller. Appli-

cations of the stand-alone controller could include cluster controllers, peripherals managers, line concentrators or any other small system.

V. THROUGHPUT ANALYSIS

This section of the application note deals with studies that have been done to quantify the performance of the iSBC 544 board in both the stand-alone and intelligent slave modes. After describing the various test configurations and assumptions the data will be presented in graphical form and analyzed. The graphical data can be found in Appendix C.

Stand Alone Throughput

The first two tests were run to determine the absolute best case throughput of the iSBC 544 board configured as a stand-alone computer. Figure 9a shows the iSBC 544 controller continuously outputting data from four buffers to the four USARTs. Figure 9b shows essentially the same setup with eight channels, four on the iSBC 544 board and four on the iSBC 534 expansion card. In each configuration the 8251A was run in synchronous mode and the baud rate was incremented until the transmitter empty signal from the USARTs became active. Further increments of the baud rates would not have resulted in higher throughput since the CPU was already spending 100% of its available time responding to USART service requests.

The maximum rate for the first configuration (iSBC 544 board only) was 32,311 baud per channel. When the iSBC 534 expansion board was added a rate of 12,186 baud per channel was achieved. The drop in baud rate was due to the extra processing required by the offboard logic (eg. reading 8259 interrupt controller on the iSBC 534 board to determine which device is requesting service).

It should be noted that the serial throughput tests were run with almost no overhead and no actual processing of the data involved. The reader is expected to apply information on the amount of overhead expected in each individual application. For instance, if the application code for a given system is expected to utilize approximately 40% of the available CPU time and we wish to run four
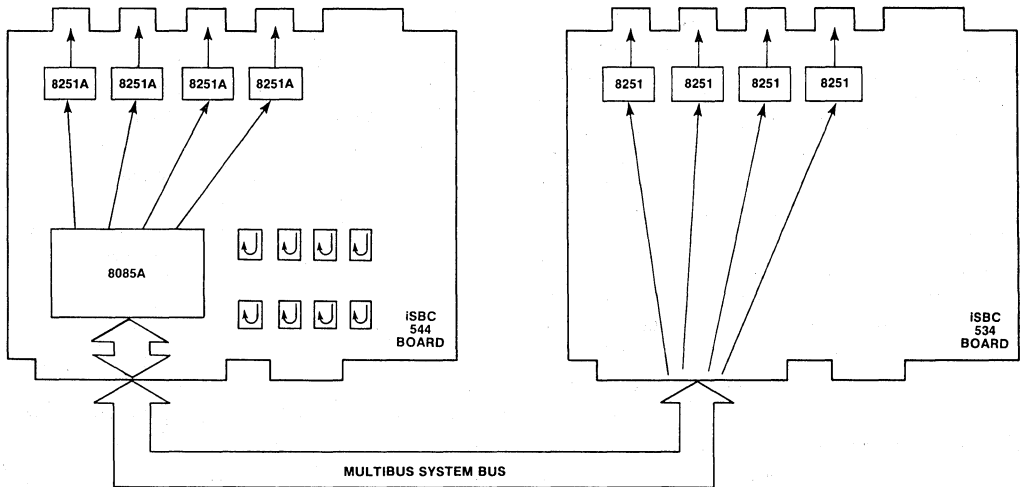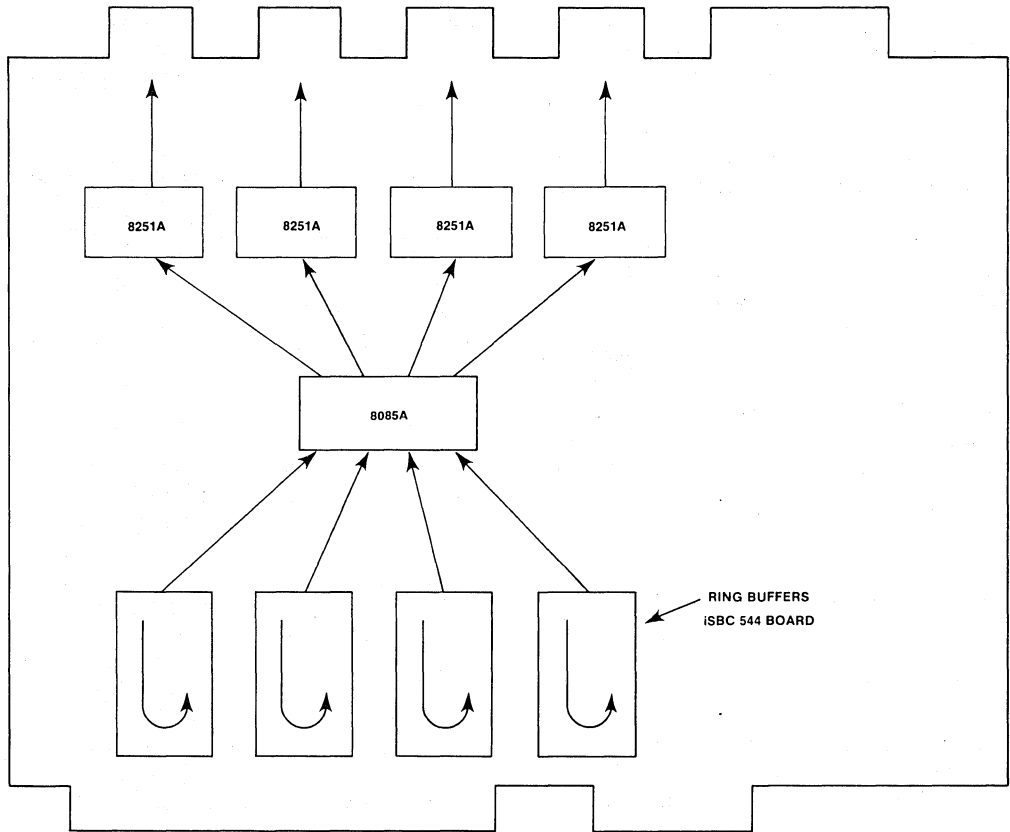
Figure 9a and 9b. Stand-Alone Throughput Configurations

Labels in figure:

8251A (×4, top group)

8085A

RING BUFFERS
iSBC 544 BOARD

8251A (×4, lower left)

8085A

iSBC 544 BOARD

8251 (×4, lower right)

iSBC 534 BOARD

MULTIBUS SYSTEM BUS

full duplex channels in asynchronous mode the estimate of maximum baud rate would take the following form.

32,331 baud per channel — 40% = 19,398.6 baud

19,398.6 baud per channel synchronous x 10/8 = 24,248.25 baud asynchronous

24,248.25 baud per channel half duplex/2 = 12,124.125 full duplex

Therefore, the maximum standard baud rate would be 9600 baud per channel in full duplex asynchronous mode.

Intelligent Slave Throughput

The remaining four configurations were set up to determine the effectiveness of the intelligent slave in the overall system. The general system configuration is illustrated in Figure 10. The boards surrounded by the box represent the systems under test. The disk controller and two iSBC 80/20 single board computers were active on the bus to simulate the normal bus traffic load in an application system. Various bus duty cycles were created using the computers and the disk controller to perform tasks that resulted in fixed bus utilization.

reflected the amount of bus time, master CPU time and slave CPU time left available to applications oriented tasks. In each case this percentage of time available was measured as the baud rate was stepped up so that a graph could be constructed showing time available as a function of transmission speed.

CPU free time was measured using a counting program running in the background. After each USART interrupt the counter was started. As interrupts from other sources came in the counting was preempted and then resumed after servicing the interrupt. When the next USART interrupt occured, the counter contents were examined and if the value was lower than the stored value the current value became the stored value. After ten minutes the stored value was retrieved and used as an indicator of the worst case time available between interrupts.

System bus utilization was measured using the circuit shown in Figure 11. The voltage measured by the digital voltmeter represented a time average of the voltage at the output of the flip-flop. A calibration chart was created using a pulse generator to simulate various duty cycles and then this chart was used to measure bus activity while the test was running.

Figure 10. General System Configuration for Throughput Testing
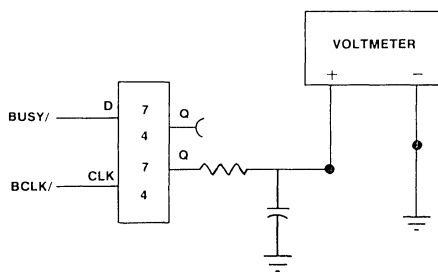
Figure 11. Bus Free Time Measurement Circuit

In each configuration a single full duplex channel was set up with the input provided by another CPU. Only those functions dealing with system overhead were included and the data measured

Configuration 1 is shown in Figure 12. This system uses a typical method of communications expansion with the iSBC 80/30 single board computer handling the lines directly via the serial I/O ports on the iSBC 534 I/O controller board.
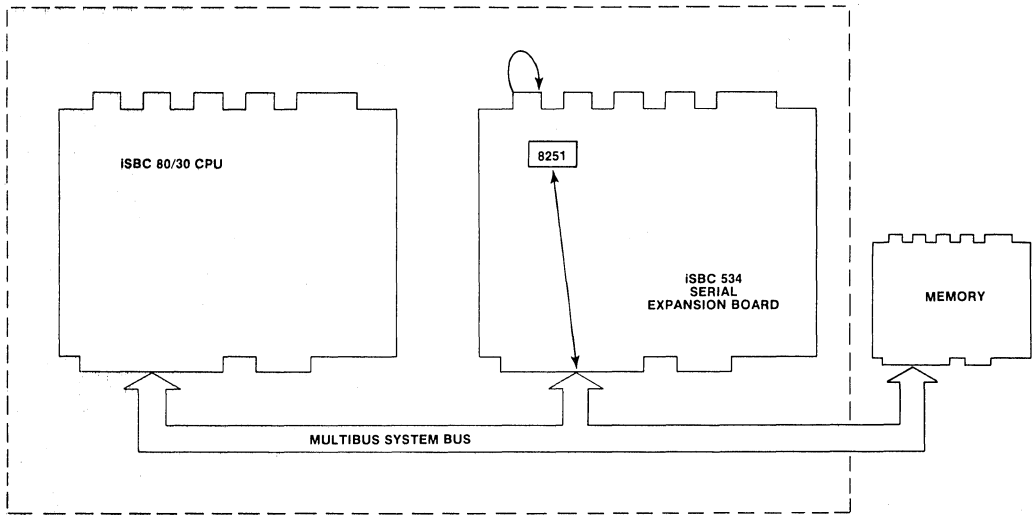
Figure 12. System Throughput Test, Configuration 1

The second configuration (Figure 13) illustrates the performance of the traditional DMA controller approach. If the communications controller had DMA logic instead of a dual port memory and transferred data directly into system memory the performance would be as observed in this test.

In configuration 3 (Figure 14) the iSBC 544 board was used in the intelligent slave mode. This configuration differs from the second in that memory transfers involved only local memory and bus access was not required on a per character basis.

The fourth and final configuration sought to identify the loading that additional intelligent slave controllers would impose on master CPU time and bus free time. Figure 15 shows the
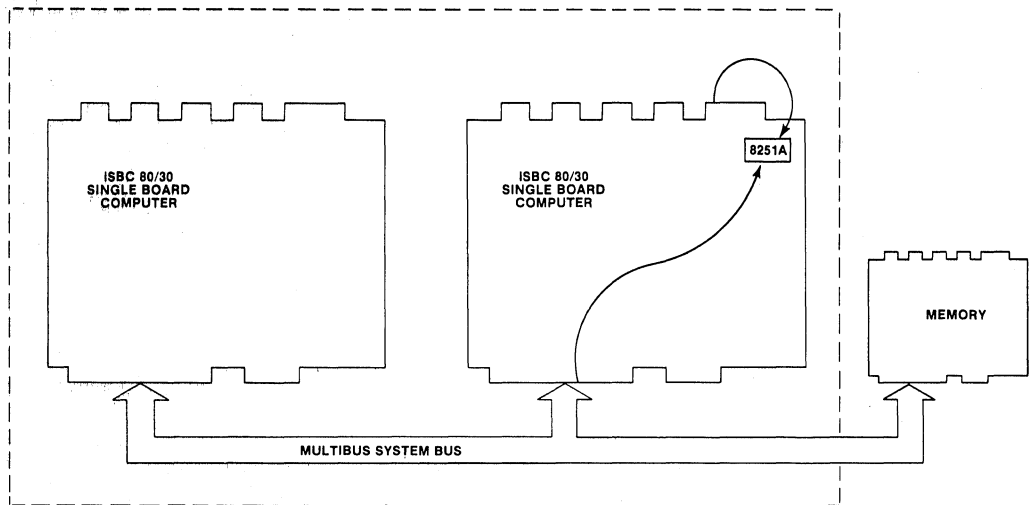
Figure 13. System Throughput Test, Configuration 2
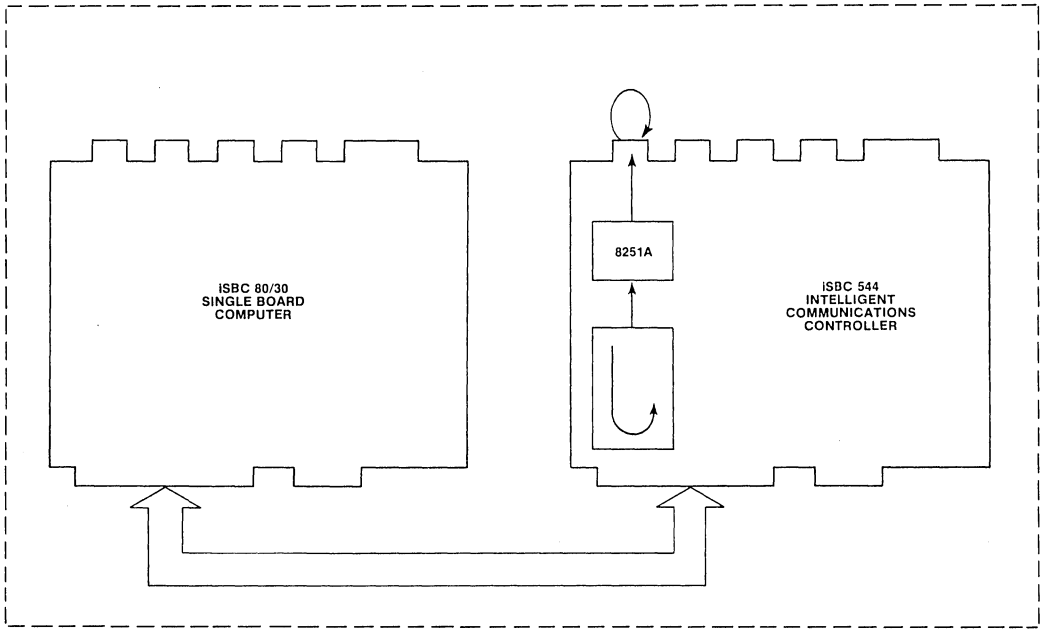
Figure 14. System Throughput Test, Configuration 3
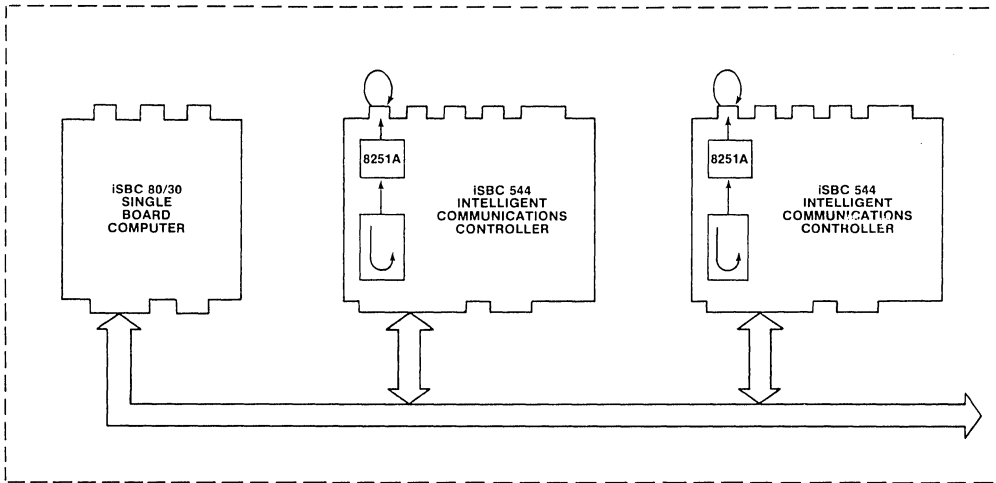
Figure 15. System Throughput Test, Configuration 4

configuration with two iSBC 544 boards executing identical programs.

The graphical presentation of the results is split into two sections. The first three graphs (Graph 1 through Graph 3) show the relationship between baud rates and the master CPU, system bus, and slave CPU utilization. All of these results are based upon tests with 30% induced bus traffic (i.e., the two iSBC 80/20 computers and the iSBC 204 disk controller were active.)

In graph 4, processor free time is graphed as a function of bus traffic. The processor in this case is the one actually involved with the data on a per character basis (i.e., iSBC 80/30 board in configuration 1, iSBC 80/30 board simulating DMA Controller in configuration 2, and iSBC 544 board in configuration 3).

Finally, graph 5 illustrates the maximum attainable baud rate for each configuration as the bus traffic is increased.

All of the graphs identify the relative performance difference between the configurations. Absolute numbers are not presented due to the fact that the overhead imposed by the test software affects the CPU time being measured. Since the overhead applies equally to all configurations, the relative performance indications are valid.

Based upon the data presented, the DMA controller and intelligent slave use 3 times less CPU time than an I/O controller. Also, the iSBC 544 intelligent slave generates 12% and 6% less bus traffic than the I/O controller and DMA controller respectively. Finally, the intelligent slave uses 8% less slave CPU time than the DMA controller approach.

The earlier discussion that dealt with the intelligent slave architecture pointed out that the distribution of intelligence would offload the master CPU so that it would retain sufficient processing power for the actual application, whatever that may be. In addition, it was stated that the assumption of the slave role would relieve the slave CPU of system overhead and at the same time reduce system bus traffic. All of these assumptions are supported by the results of the testing presented here.

The second set of graphs identify the effects of bus traffic on the performance of the various components of the system. The main observation to be made in this sequence is the drop in CPU free times and maximum baud rates that occurs when the bus gets busy. This effect is observable in the communications processor free time when the iSBC 534 expansion board or the DMA controller configuration is used. No effect is evident in the configuration with an iSBC 544 board.

The cause of this effect is the amount of bus access required by each configuration to move the characters from the USART to or from the buffer. With an iSBC 534 board the master CPU receives an interrupt, polls the offboard 8259 interrupt controller, reads in a character, stores it in system memory and sends an end of interrupt command to the offboard interrupt controller. When the iSBC 80/30 computer receives an interrupt all processing is performed onboard until a bus access is required to move the data byte from/to memory. In the case of the intelligent slave, all processing for a character is performed onboard. Thus, as the system bus becomes very fully utilized, the delays encountered in receiving bus access by the first two configurations become significant.

The fourth configuration, which was set up to test the effects of adding more intelligent slaves, shows that extra slaves cause no appreciable increase in system load. All of the data points for two slaves were identical to the points for one slave in graphs 1 through 5.

VI. APPLICATION EXAMPLES

A Distributed Control System

The potential applications for a product like the iSBC 544 communications controller are almost unlimited and not restricted to the traditional Data Communications market. The first application example that is studied concerns industrial automation. Due to the fact that the system is distributed and requires a generalized network, the iSBC 544 board is a natural prospect to handle the communication links between the various nodes in the system.

Design Requirements

The system to be designed is intended to provide the framework for a family of distributed control systems where the configurations and the objects to be controlled vary from system to system. Figure 16 shows the general picture of the system.
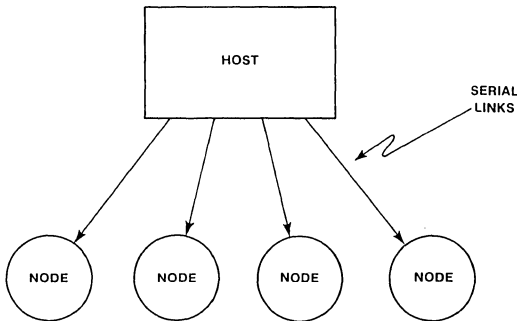
Figure 16. General Diagram of Distributed Control System

The host is responsible for providing supervisory control and a high-level human interface. The system can be expanded as shown in Figure 17 where the controllers attached to the host are replaced by intermediate nodes which contain controllers or other nodes. This process can be continued as far as is necessary to provide the needed number of controllers. Each controller in the diagram represents a localized closed loop control system that is tailored to the specific application.

The following system requirements need to be met by the computer network:

- The host CPU must have sufficient computational power to handle the human interface, mass storage management, supervisory control calculations and network control.

- The host CPU must not be overly burdened by low-level communications functions if it is to handle the other duties assigned to it.

- Node controllers must be capable of handling 8 medium speed lines and also modems and autocall units since the nodes or controllers attached may be remote.
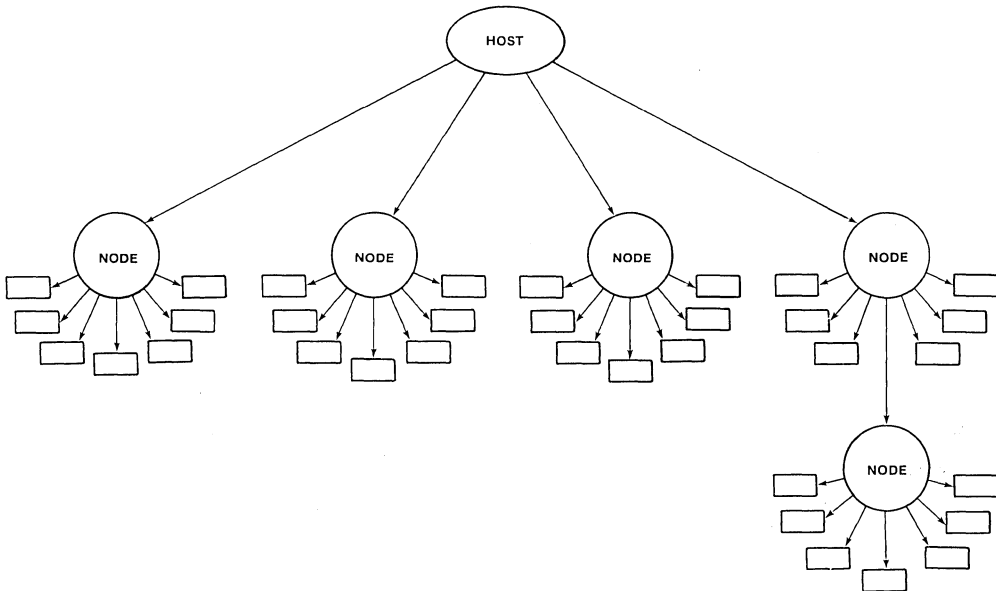
Figure 17. Expanded Diagram of Distributed Control System

- The message transmission format must be independent of the configuration and end application. The nodes in the network must be capable of passing through messages with and without interpreting the contained data.

- The system must be capable of auto-configuration (since the network configuration is tailored to the specific application, the host must be able to automatically determine the setup at power on).

- Each node controller is responsible for verifying the integrity of the nodes attached.

System Configuration

Based upon the design criteria and the benchmark information the chosen configuration uses an iSBC 86/12 Single Board Computer as the host with an iSBC 544 intelligent slave handling the communications load for the CPU. The USART on the CPU board will talk to the local terminal and an iSBC 206 Hard Disk Controller will be used to provide up to 40 Megabytes of mass storage capacity.

The requirements for the node controllers point to an iSBC 544 board configured as a stand-alone communications computer with an iSBC 534 board as expansion to provide the necessary 8 lines. The throughput data indicated a raw throughput value of 12K baud on each channel. With the data rates expected being far below this, sufficient time will be left over for background functions. Thus, the software requirements for each node can all be met by the CPU on the iSBC 544 board and the inclusion of an expansion board does not necessitate another iSBC computer.

A typical controller in the system would look like that shown in Figure 18. The iSBC CPU handles the local closed loop control, using parametric information sent from the host. This information would typically include setpoints, tolerances and alarm limits. The serial channel on the CPU will be used to maintain the link to the next level in the network.

Preliminary Design

The message format that the system uses is shown in Figure 19. When multiple nested levels
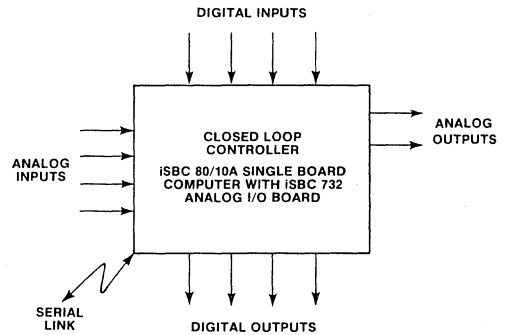
Figure 18. Typical Controller in Distributed System

Figure 19. Message Format
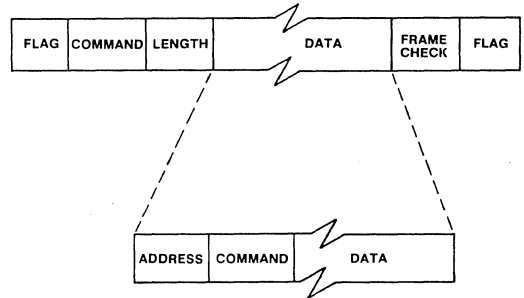
Figure 20. Nested Level Address Information

of nodes are used the data area of the message contains command and address information for the next level down (Figure 20). Interpretation of the commands in a given message is done on an individual basis except for a set of system-wide commands (eg. IDENTIFY is a system command meaning respond with your ID code). The flexibility afforded by this scheme can be extremely useful in a system where the end applications and configurations may be quite diverse (eg. a node controller that is processing a transmit command may be the only one that knows that it is sending to another node via a phone line and thus it interprets the contained data differently than another node would). The level of intelligence and the ease of programming of the iSBC 544 board make this generalized transmission scheme possible.

The simplest means of auto-configuration requires each controller in the system to send an identity message to the nearest node. This node would know the logical address of the controller that sent the message and would attach this address to the message and retransmit it to the next level as illustrated in Figure 21. This process would be repeated until the host is reached and would contain, at this point, all necessary address information to reach the given controller.
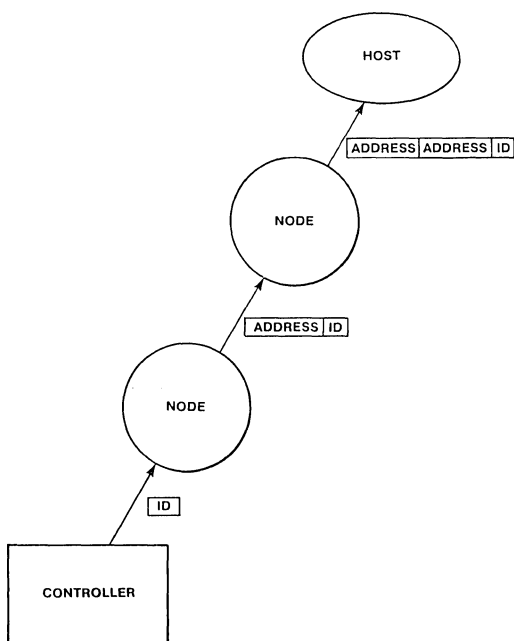
Figure 21. Auto Configuration

The human interface on the host would provide a mapping mechanism to attach meaningful symbolic names to the various nodes in the system. This labeling, along with the application specific control algorithms, make it possible to say something like "lower the temperature on the third floor to 68°F". The host breaks this information down into setpoints and tolerances, uses the map to determine the path to the node(s) responsible for the third floor and transmits the information through the network.

Each node controller in the system has the added responsibility of verifying the integrity of all the nodes attached to it. This duty can be handled by periodic background commands issued from the host and propagated through the network. Each node is responsible for passing the command along and also polling the nodes attached to it and reporting back any error conditions.

Summary

Through the use of a powerful 16–bit iSBC Single Board Computer, various low-cost 8–bit iSBC CPUs and the iSBC 544 communications controller, a flexible and extensible distributed control system is easy to design. The dual nature of the iSBC 544 board provides both an intelligent front end to the host computer and a high-speed stand-along nodal concentrator. The ability to individually customize the software on each controller provides for an easily expandable system design.

Terminal Cluster Controller

The second application example concerns itself with a terminal cluster controller. The system shown in Figure 22 uses a number of "dumb" terminals and makes them appear "intelligent" via a local microcomputer system. The local microcomputer interfaces with the operator and accesses a local data base to provide an inquiry and data entry service. When necessary, the local microcomputer is capable of calling the host via an autocall unit and exchanging information and updates to the data base.

Design Criteria

The terminal cluster controller must meet the following criteria:

● Support must be provided for from four to sixteen operator terminals all running at rates up to 2400 baud.

● Line editing on input must be provided (delete characters, delete lines and pause output).
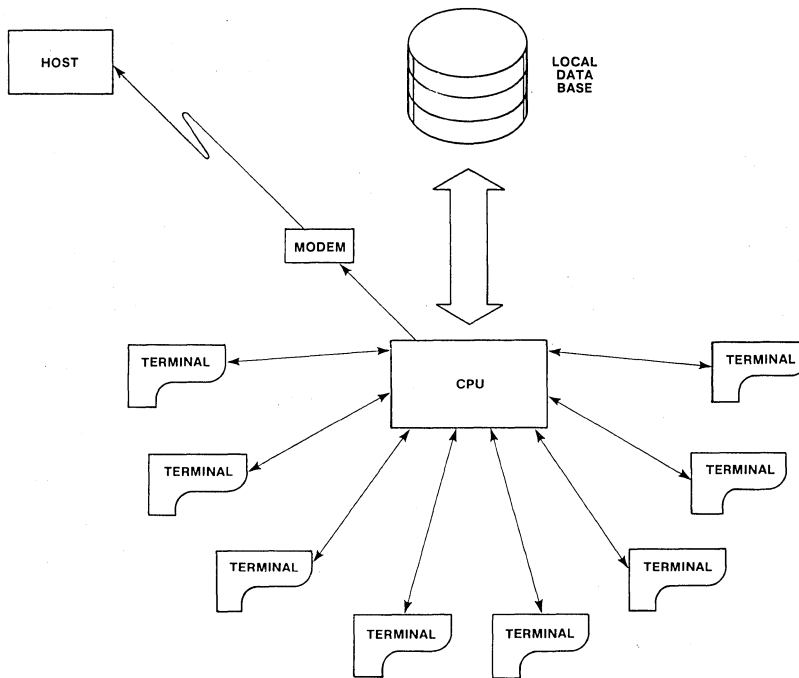
Figure 22. Terminal Cluster

- Support for the terminals must be configurable in that certain stations may require different screen formats.
- Support for an optional hard copy device must be allowed for.
- A considerable amount of CPU free time must be available after the basic terminal facilities are included. This is due to the fact that the data base management software to be written to run on the master single board computer will be extensive.
- Type ahead would be a desired feature since the processing on the master CPU after a line of input has been transmitted may cause a delay in responding and we would like to have the ability to continue entering input while waiting for the response.

System Configuration

The specific iSBC products needed to implement the system described are the iSBC 80/30 Single Board Computer with an iSBC 032 RAM Expan-

sion Board, an iSBC 206 Hard Disk Controller and one to four iSBC 544 Intelligent Communications Expansion boards. Intel's RMX/80 Real-Time Multitasking Executive will provide the basis for the software system and will include disk file support for the iSBC 206 controller through DFS/80. The full system configuration is illustrated in Figure 23.

BLOCK DIAGRAM
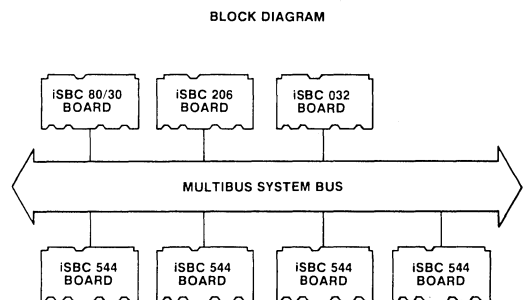
Figure 23. Terminal Cluster Controller System Configuration

Preliminary Design

The first design decision to be made involves the distribution of system functions. Due to the requirements for line-editing and type-ahead the software for processing characters input from the terminal keyboards will be somewhat lengthy. The standard terminal output handler will be very small but provisions for special screen format controls and/or hard copy devices must be allowed for. All of these requirements lead to the use of the iSBC 544 controller for all terminal functions. If the master CPU were burdened with all of these duties it would be unable to adequately perform its data base management functions. The fast CPU and 8K PROM capacity of the iSBC 544 board will be more than adequate for the task at hand.

The throughput tests indicate that the loading imposed by expanding the number of terminals (and therefore the number of iSBC 544 boards) will not adversely affect the performance of the rest of the system. Master CPU free time and bus traffic data for two intelligent slaves in the system were identical to the numbers for one slave. Thus, since the iSBC 80/30 single board computer and the MULTIBUS system bus can handle one iSBC 544 controller they can also handle the maximum of four controllers that may be required by this application. The only observable effect will be caused by the load the extra operators impose on the data base software itself.

The software needed for the iSBC 544 board is now defined and divided into three major pieces; a terminal input handler, a terminal output handler and system software to support the handlers. Since the input and output handlers are invoked via USART interrupts, all that need be done is to write a single routine for each handler and have it talk to all of the devices on the board. This can be accomplished by vectoring the proper interrupts to the entry point of the routine and then polling the 8259A interrupt controller to determine which device needs servicing.

The standard terminal input handler needs to read in the available character from the USART,

check it to see if it is a special command character and, if not, store it into a buffer. If a command character is encountered, the handler will respond by performing the appropriate operation.

The standard terminal output handler simply takes characters out of a buffer upon interrupt from the transmitter and sends them to the appropriate USART. If a different output handler needs to be substituted for a special terminal or a hard copy device, a new routine can be included by modifying the interrupt vector address in the 8259A jump table.

Since the RMX/80 Real-Time Multitasking Executive is being utilized on the master CPU it is desirable to create an RMX/80 handler for the iSBC 544 boards that accepts and processes normal terminal handler request messages. In this manner, application tasks that formerly communicated with the on-board USART via the RMX/80 Terminal Handler can be made to talk to one of the devices on the iSBC 544 board by simply changing the address of an exchange. The following paragraphs, as well as paragraphs in the section on system software, assume a knowledge of the RMX/80 Real-Time Executive. This knowledge is not necessary to use the information contained in this application note. Interested readers are referred to the RMX/80 references listed in the front-piece.

Since this application can have from one to four iSBC 544 boards the RMX/80 driver will need to be configurable. A set of tasks and exchanges will be created for each terminal in the system. One task and exchange pair will accept and process terminal input request messages while another pair will process terminal output requests.

The remaining piece of software that is needed by this system will provide the means for getting commands and data between the master and intelligent slave. Since this is a common need in any system utilizing an intelligent slave we will develop a general purpose scheme that can be used by any application. In this manner, a routine such as the terminal input handler can be written without any concern for how it will get the data it is inputting to the master CPU; all it need do is call upon a standard routine to "transmit"

the data. With these thoughts in mind, the following section discusses the system software developed for master-intelligent slave communication. After the discussion of the system software we will revisit the software for the second application as an example of the use of the data transfer routines.

VII. SYSTEM SOFTWARE

In the earlier discussion of master-slave protocols, the notion was presented of developing a general purpose data transfer scheme which would enable the applications routines on both the slave and master to operate without concerning themselves with protocols and synchronization. This scheme can be implemented by designing a set of primitive routines to handle the data transfer activities. Thus, Figure 8b is expanded as shown in Figure 24 and the applications processes now call upon the primitives to handle the communications between the master and the slave.

Data Transfer Primitives

The basic mechanism used by this implementation of the primitives is a wraparound queue as shown in Figure 25. Each 8251A device has associated with it, in dual port memory, an input and an output queue each of which have a *give*
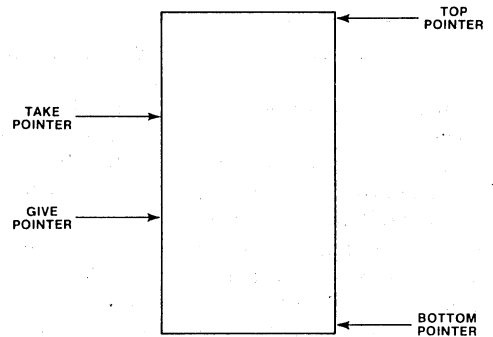
Figure 25. Wrap-around Queue Used by Data Transfer Primitives

and a *take* pointer. The *give* pointer contains the address of the next location in the queue that is available for filling with data. The *take* pointer contains the address of the next byte in an output queue that has been filled and is available. A queue is empty when the *give* and *take* pointers are equal and it is full when the act of incrementing the *give* pointer would make it equal to the *take* pointer. A *wrap* function is defined to increment a pointer such that an increment past the bottom of the queue "wraps" the pointer around to the top of the queue.
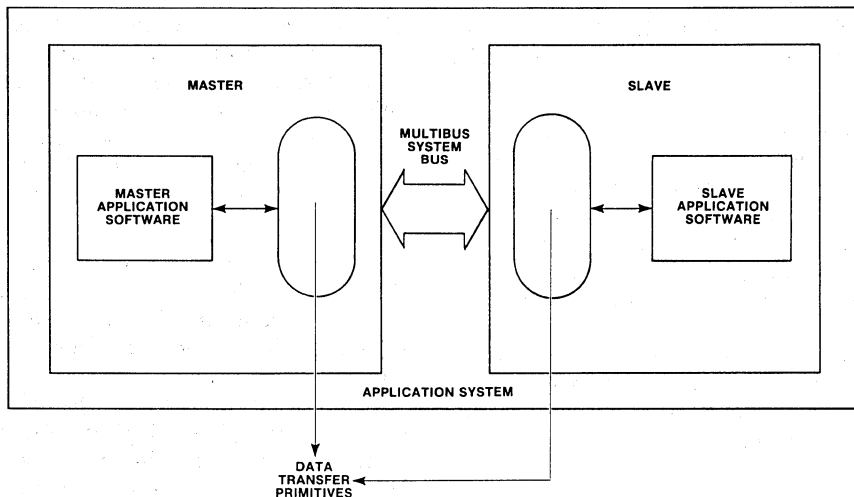
Figure 24. System Software Diagram with Data Transfer Primitives

The primitives all make use of a queue information block located at the base address of the slave's dual port memory (Figure 26). All pointer information is base relative to accommodate the needs of the two CPUs who have different memory maps. The two flag bytes carry information for master-slave and slave-master synchronization signals.
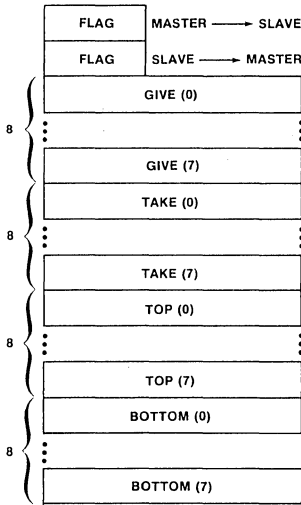
Figure 26. Queue Information Block

The set of primitives provides two distinct methods of information transfer, line oriented and byte oriented. The line oriented primitives are listed in Table 1. Both *get$line* and *send$line* transfer information between the queues and buffers provided by the caller. The disadvantage of this scheme is the number of memory moves needed to transfer information. The advantages of the line oriented method are the relative efficiencies and the simplicity of the interface from the calling routine.

The byte oriented primitives (Table 2) allow the calling routine to transfer data directly into and out of the queues. An example of the sequence for putting a character into a queue is illustrated in Figure 27. The routine servicing the receiver ready interrupt calls *next$space* to get a pointer to the next available slot in the queue and then uses this pointer to transfer the data byte directly into the queue. The *new$line, xmit, open$line* and *receive* primitives are necessary since the global *give* and *take* pointers cannot be modified until all manipulations on the affected section of the queue are complete. If the pointers were modified continuously the routine gathering the data from the other side may see invalid data.
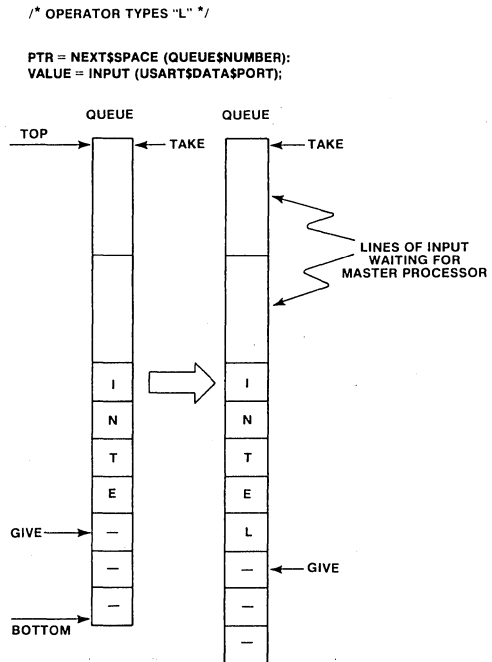
Figure 27. Sequence for Putting Data Into Queue

Table 1
Line Oriented Primitives

Primitive	Arguments	Usage
send$line	Queue$token, buf$ptr, *count* Returns: overflow	Inserts *count* characters into queue from buffer If insufficient room available, overflow indicates how many would not fit
get$line	Queue$token, buf$ptr, count Returns: Actual	Retrieves count characters from queue and puts them in buffer Actual indicates how many were actually moved

The remaining primitive routines deal with the general purpose needs of the application software with regard to interrupts, initialization and status checking. A full list of these support routines is contained in Table 3.

There are many features of this implementation and a few of them should be pointed out at this time. By defining a general purpose set of primitive routines to handle the data transfer, the actual means by which the bytes are transferred between slave and master is not visible to the calling routine. If the actual mechanism used needs to be altered the change will not affect the application software as long as the same external interface is maintained.

Another important feature of the primitive routines is the fact that they do not interpret the bytes that are sent to them. Due to this fact, the applications routines are free to send commands and parameters interspersed with the actual data. As an example, the terminal driver on an iSBC 544 board might perform format control based upon table information. The master applications software could use the data transfer primitives to transmit commands and parameters to the slave to update its format control information. Another advantage of the fact that the data is not interpreted is that it allows the calling routine to determine what data gets sent along. For instance, a specific terminal might be transmitting ASCII code while the master

Table 2

Byte Oriented Primitives

Primitive	Arguments	Usage
new$line	Queue$token Returns: ptr	Sets up a queue for byte oriented input. Ptr returned points to the first available byte.
next$space	Queue$token Returns: ptr	Increments the temporary give pointer to the next open space. Ptr returned either points to next byte or is zero specifying full queue.
back$space	Queue$token Returns: ptr	Decrements temporary give pointer. Ptr returned either points to byte or is unchanged indicating that the global give pointer was reached.
xmit	Queue$token Returns: status	Closes off a line entered via byte mode by updating global give ptr to equal temporary give ptr. Status is either "normal" or "null".
open$line	Queue$token Returns: ptr	Opens up a line for byte oriented output. Ptr returned either points to the next byte or is zero indicating an empty queue.
next$char	Queue$token Returns: ptr	Increments temporary take pointer. Ptr returned either points to next byte or is zero indicating an empty queue.
receive	Queue$token Returns: status	Closes off a line retrieved in byte mode by updating global take pointer to equal temporary ptr. Status is either "normal" or "null".

Table 3

Support Routines

Primitive	Arguments	Usage
get$status	Queue$token Returns: status	Returns status of queue. Possible values are "normal", "empty", "full" and "null".
set$interrupt	Queue$token, type Returns: status	Generates a slave → master or master → slave interrupt. Type code 0 is illegal and codes 8H → 0FH are reserved for use by the primitives.
set$handler	Queue$token, handler$adr Returns: status	Inserts address into vector table used for handling interrupts described above.
s$init	none	Called from slave software to initialize.
m$init	none	Called from master software to initialize.

software is expecting EBCDIC. The routine on the slave can very easily perform the necessary code conversion before stuffing the data into a queue.

Sample Slave Software

Given the existence of the primitive routines the applications routines on the slave and master can deal with the specific duties of each device. The following paragraphs revisit the code from application example 2, first for the slave and then for the master. Full code listings for these programs can be found in Appendix D.

The flowchart for the terminal input handler resident on the iSBC 544 board is shown in Figure 28. Support is provided for deleting characters (Rubout), deleting lines (control-X), pausing and resuming output (control-S and control-Q) and terminating lines (escape and carriage return). The sections of code reproduced below use this terminal input handler to present an example of the use of the data transfer primitives to enter and edit a line of input from a terminal. The byte variable *value* is based on the address variable *value$ptr* which is assigned by calls to the primitives. The routine *var$inp* inputs and returns a data byte from an I/O port specified by a calling parameter. This is necessary since the particular USART to be serviced is determined by reading the 8259A in-service register.

```
/* case 1; rubout; delete char */

do;
    new$ptr=back$space(token);
    if new$ptr=length$ptr then
        dummy=echo(token+1,.(bell),1);
    else
        do;
            dummy=echo(token+1,.(BS,SP,BS),3);
            ptr=new$ptr;
            count=count-1;
        end;
end;
```

Following this, the byte input is checked to see if it is a control character and if so a block within a DO CASE statement is executed. As an example of one of these blocks, if the character input was a RUBOUT the code sequence below is executed. The *back$space* primitive is called and a tempo- rary pointer is returned to a location in the queue. A check is made to determine if the line was empty and, if so, a bell is echoed to signal the operator. If the pointer returned did not indicate

an invalid RUBOUT the real pointer is assigned the value of the temporary pointer and a back- space, space, backspace is echoed to delete the previous character on the screen. Lastly, the character count for the current line is decre- mented.

```
VALUE$PTR=NEXT$SPACE(QUEUE$NUMBER);
VALUE=VAR$INP(USART$DATA$PORT(NUM));
```
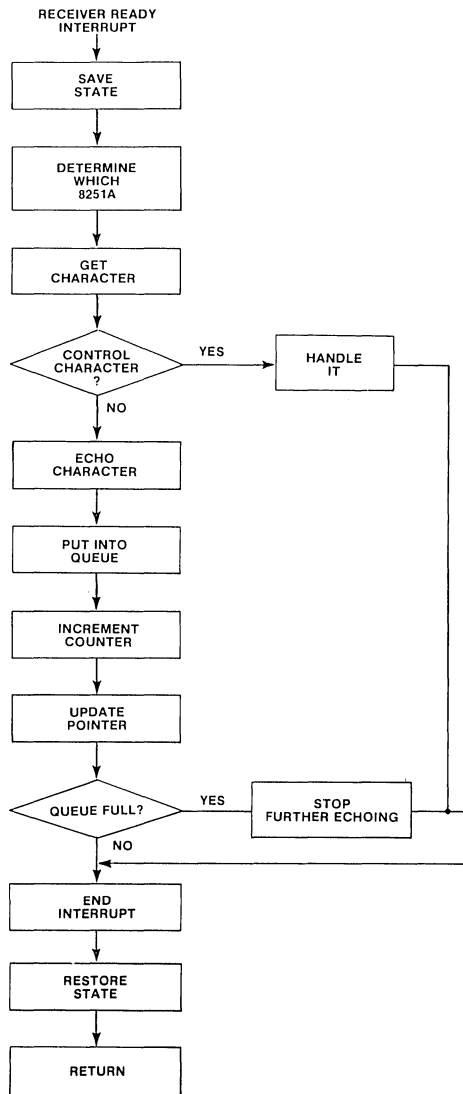

Figure 28. Flow Chart for Terminal Input Handler

In order to facilitate retrieval of the proper amount of information on the master side, the first byte of each message is defined to contain the number of characters in the message. Thus, when the master routine needs a line of input he uses the first byte as a count to retrieve the full line. The requirement for type-ahead is met by this mechanism since the number of lines in the queue at a given time is limited only by the length of the queue. When a full line of input is finished, the terminal input handler generates a slave to master interrupt to signal the master routine who may be waiting for this event.

The flowchart for a minimal terminal output handler is shown in Figure 29. Upon receipt of a
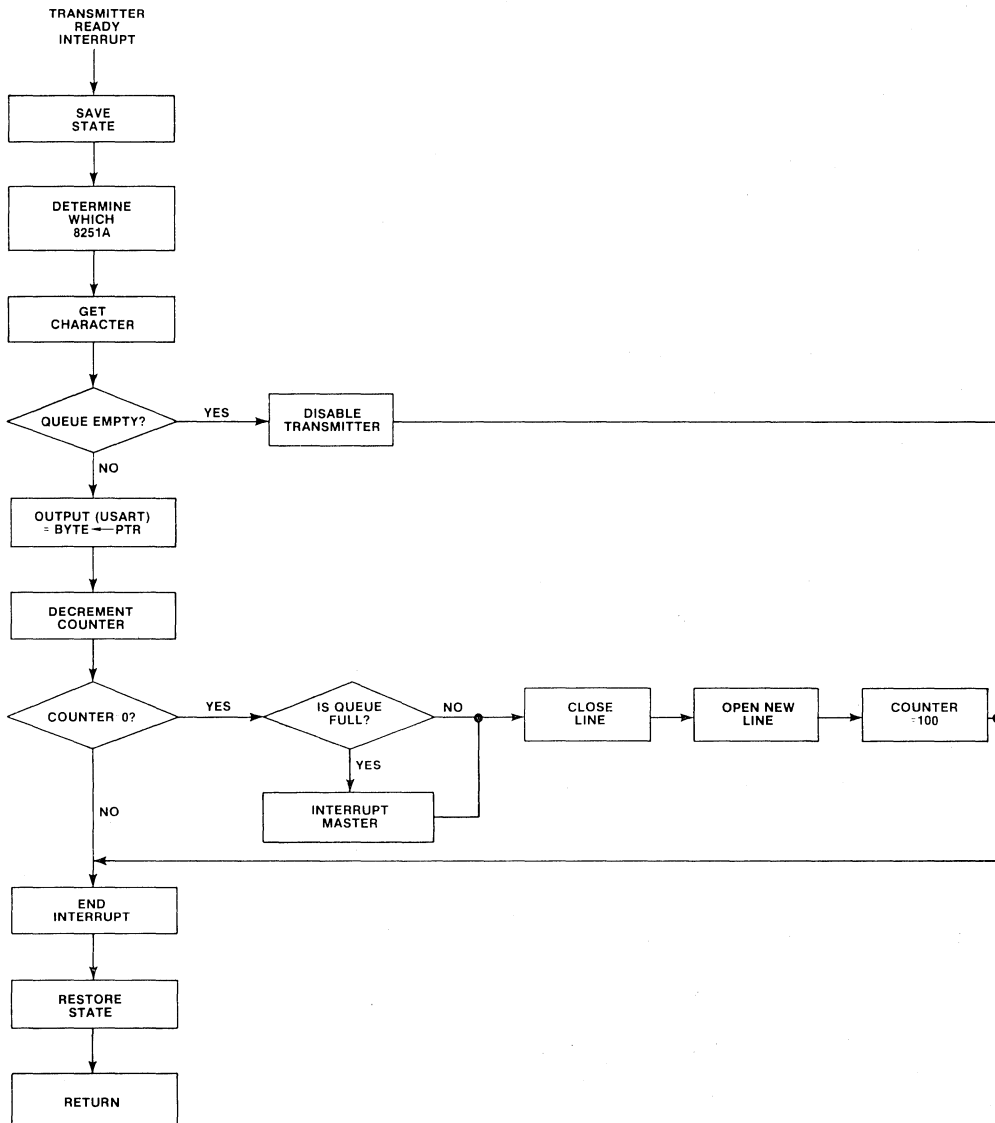
Figure 29. Flow Chart for Terminal Output Handler

transmitter ready interrupt the output handler requests a character from the appropriate queue. If one is available it is output to the USART. If the queue is empty, the transmitter is disabled. Whenever the master routine sends a line into the queue it will generate an interrupt to signal the slave handler and the transmitter will be reenabled. A line is opened via a call to *open$line* and it is kept open until 100 characters have been retrieved via calls to *next$byte*. At this time the line is closed by a call to *receive* making the space available to be reused. After this, a new call to *open$line* starts the process over again. If the call to *get$status* shows that the queue was full prior to the call to receive, an interrupt is sent to the master to reawaken any routine that may have been waiting for room in the queue to become available.

Sample Master Software

The RMX/80 handler for the master single board computer that will communicate with the software on the iSBC 544 board is diagrammed in Figure 30. In addition, the RMX/80 message used to convey information to the handler is shown on the right. The full software diagram is illustrated in Figure 31.

The input driver tasks execute a reentrant routine that services a request exchange that is specified in an initialization block that is unique to each of the input tasks. The necessary information is extracted from the request message and the *get$line* primitive is called upon to get a line of input from the queue. If the call to *get$line* for the length byte is unsuccessful the input task waits at
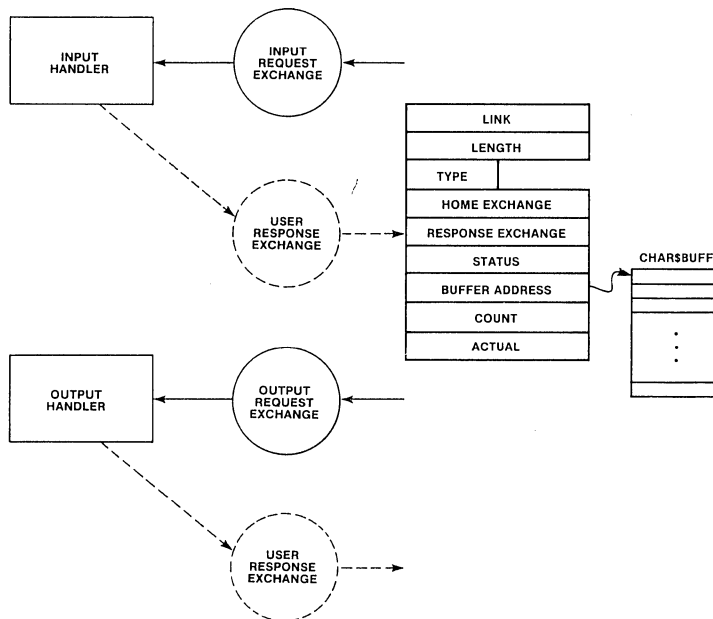
Figure 30. RMX/80 Handler for iSBC 544 Board
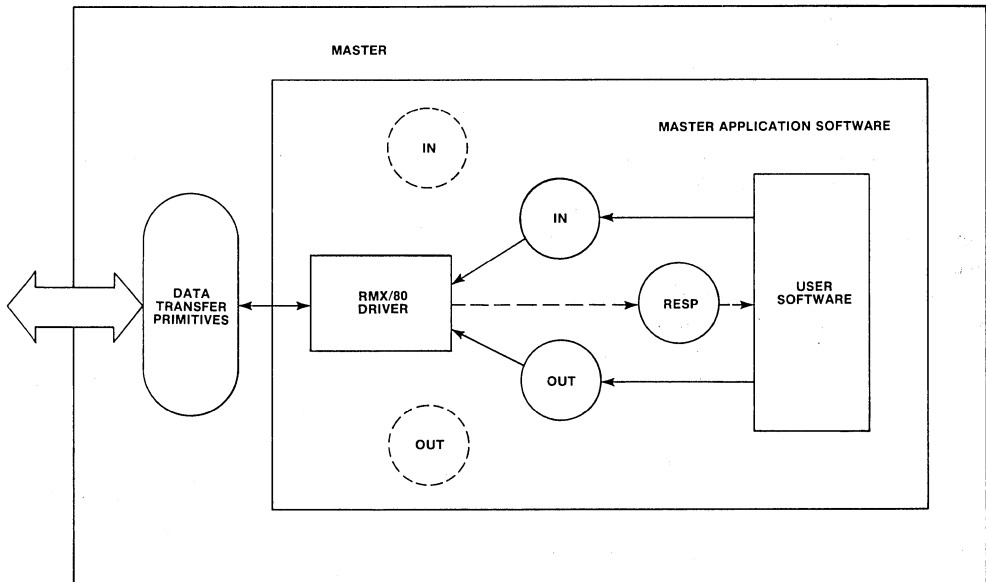
Figure 31. Master Software with RMX/80 Handler

the appropriate signal exchange for an interrupt from the slave indicating that a line is now available. Once the request is fulfilled the actual and status fields are set and the message is sent to the response exchange specified by the user.

The output handler performs in a manner very similar to the input handler. Upon receipt of a request message the handler attempts to transfer the characters from the user buffer to the appropriate queue. If the attempt is unsuccessful (ie. the queue has insufficient room available) the handler sends as many characters as will fit (count − overflow) and then waits for an interrupt from the slave indicating that room has been made available. This process is repeated until all of the data has been transmitted. As soon as the operation is complete the status field is cleared and the message is returned to the user specified response exchange.

Since the number of iSBC 544 slaves in the system is variable as are the memory base address, device programming information and

queue sizes, some means of providing configuration information to the RMX/80 handlers is needed. This information resides in the *memory$allocation$module*. Public variables are declared in this module that are used by the RMX/80 tasks to determine how many devices (and therefore how many tasks need to be created) are in the system and where in the system address space their dual port RAM is located. In addition, queue sizes and device programming information are specified here.

VIII. SUMMARY

The intent of this application note has been to introduce the reader to the concept of the intelligent slave architecture and show the versatility of the first product based upon this architecture, the iSBC 544 Intelligent Communications Controller. The hardware and software aspects of the device were studied and results of benchmark tests were presented and studied. Finally, two example applications were worked out using the product as both a stand-alone controller and as an intelligent slave.

The bottom line is that the iSBC 544 controller, due to the advanced architecture around which it is designed, can be the means to the end for any application that requires communication. The dual nature of the controller provides the full power of a single board computer to the small application while the large system can make use of the fully programmabale intelligent slave to free the CPU for complicated processing duties.

Figure A-1. iSBC 544 Input/Output and Interrupt Block Diagram

Figure A-2. iSBC 544 Memory Block Diagram

APPENDIX B

ISIS-II 8Ø8Ø/8Ø85 MACRO ASSEMBLER, V3.Ø   MODULE   PAGE   1

```
 LOC  OBJ          LINE          SOURCE STATEMENT

                      1 $MOD85
ØØ8Ø                  2 BASE      EQU    8ØH              ;BASE ADDRESS OF 204
ØØE4                  3 MMSET     EQU    ØE4H             ;MASTER MODE SET ADDRESS
ØØE5                  4 MMRSET    EQU    ØE5H             ;MASTER MODE RESET ADDRESS
ØØ52                  5 READ      EQU    52H              ;READ COMMAND CODE
4ØFF                  6 TCOUNT    EQU    4ØFFH            ;TERMINAL COUNT AND DMA MODE `OF
ØØØ4                  7 DMAMOD    EQU    Ø4H              ;DMA MODE WORD
ØØØ1                  9 TADDR     EQU    1                ;TRACK ADDRESS
ØØØ2                  9 SADDR     EQU    2                ;SECTOR ADDRESS
                     1Ø           DSEG
ØØØØ                 11 BUFFER: DS     128              ;SECTOR BUFFER
                     12         ;
                     13         ;      TEST OF CAPABILITY FOR 544 TO SHARE MULTIBUS
                     14         ;      WITH OTHER MASTERS. ROUTINE PROGRAMS THE 204
                     15         ;      BOARD, INITIATES A READ TRANSFER, WAITS FOR
                     16         ;      AN INTERRUPT AND THEN TRAPS TO ICE85 BREAK-
                     17         ;      POINT AT 2ØØ.
                     18         ;
                     19           ASEG
ØØ2C                 20           ORG    2CH              ;RST 5.5 ENTRY POINT
ØØ2C D3E4            21           OUT    MMSET            ;SET MASTER MODE
ØØ2E DB81            22           IN     BASE+1           ;GET RESULT
ØØ3Ø AF              23           XRA    A                ;SET FLAGS
ØØ31 C4390Ø      C   24           CNZ    ERRTRP           ;NON-ZERO RESULT; ERROR TRAP
ØØ34 FB              25           EI                      ;REENABLE
ØØ35 C9              26           RET                     ;CONTINUE ON
                     27         ;
                     28         ;      MAINLINE ROUTINE
                     29         ;
                     30           EXTRN  INIT24           ;204 INITIALIZATION ROUTINE
                     31           EXTRN  WAITC            ;WAIT FOR 204 NOT BUSY ROUTINE
                     32           EXTRN  WAITP            ;WAIT FOR 204 PARAMETER REGIST
                     33           CSEG
ØØØØ F3              34           DI                      ;DISABLE
ØØØ1 31FFBF          35           LXI    SP,ØBFFFH        ;SET STACK POINTER
ØØØ4 D3E4            36           OUT    MMSET            ;SET MASTER MODE FLIP FLOP
ØØØ6 CDØØØØ      E   37           CALL   INIT24           ;INITIALIZE 204
ØØØ9 3EØ4            38           MVI    A,DMAMOD         ;SET DMA MODE
ØØØB D388            39           OUT    BASE+8           ;
ØØØD 3EFF            40           MVI    A,LOW(TCOUNT)    ;SET CONTROL REGISTER
ØØØF D385            41           OUT    BASE+5           ;
ØØ11 3E4Ø            42           MVI    A,HIGH(TCOUNT)   ;
ØØ13 D385            43           OUT    BASE+5           ;
ØØ15 3EØØ        D   44           MVI    A,LOW(BUFFER)    ;OUTPUT LOW BYTE OF DMA ADDRESS
ØØ17 D384            45           OUT    BASE+4           ;
ØØ19 3EØØ        D   46           MVI    A,HIGH(BUFFER)   ;OUTPUT HIGH BYTE OF DMA ADDRESS
ØØ1B D384            47           OUT    BASE+4           ;
ØØ1D CDØØØØ      E   48           CALL   WAITC            ;
ØØ2Ø 3E52            49           MVI    A,READ           ;OUTPUT READ COMMAND
ØØ22 D38Ø            50           OUT    BASE+Ø           ;
ØØ24 CDØØØØ      E   51           CALL   WAITP            ;
ØØ27 3EØ1            52           MVI    A,TADDR          ;TRACK ADDRESS
ØØ29 D381            53           OUT    BASE+1           ;
ØØ2B CDØØØØ      E   54           CALL   WAITP            ;
ØØ2E 3EØ2            55           MVI    A,SADDR          ;SECTOR ADDRESS
ØØ3Ø D381            56           OUT    BASE+1           ;
ØØ32 D3E5            57           OUT    MMRSET           ;RESET MASTER MODE FLIP/FLOP
```

```
0034 FB            58        EI                    ;ENABLE
0035 76            59        HLT                   ;AND HALT        ; WAIT FOR INTEF
0036 C30002        60        JMP     200H          ;TRAP TO ICE85 BREAKPOINT AT 200
                   61 ERRTRP:                      ;ERROR TRAP
0039 76            62        HLT                   ;FOR NOW
                   63        END
```

PUBLIC SYMBOLS

EXTERNAL SYMBOLS
INIT24 E 0000   WAITC E 0000   WAITP E 0000

USER SYMBOLS
BASE   A 0080   BUFFER D 0000   DMAMOD A 0004   ERRTRP C 0039   INIT24 E 0000   MF
READ   A 0052   SADDR A 0002   TADDR A 0001   TCOUNT A 40FF   WAITC E 0000   WAI

A M80 :Fl:INIT24.M80

ISIS-II 8080/8085 MACRO ASSEMBLER, V3.0   MODULE   PAGE   1


```
  LOC  OBJ        LINE          SOURCE STATEMENT

                     1 $MOD85
  0080               2 BASE     EQU     80H             ;BASE ADDRESS OF 204
  00E4               3 MMSET    EQU     0E4H            ;MASTER MODE SET ADDRESS
  00E5               4 MMRSET   EQU     0E5H            ;MASTER MODE RESET ADDRESS
  0069               5 SEEK     EQU     69H             ;SEEK COMMAND
  0035               6 SPECFY   EQU     35H             ;"SPECIFY" COMMAND CODE
  0010               7 BADTR1   EQU     10H             ;SPECIFY BAD TRACKS SURFACE 1
  0018               8 BADTR2   EQU     18H             ;SPECIFY BAD TRACKS SURFACE 2
  00FF               9 NOBAD    EQU     0FFH            ;NO BAD TRACKS
  00FF              10 CTADDR   EQU     0FFH            ;CURRENT TRACK ADDRESS NOT KNOWN
  000D              11 CHARS    EQU     0DH             ;SPECIFY DRIVE CHARACTERISTICS
  0008              12 SETTLE   EQU     08H             ;HEAD SETTLE TIME(SA800)
  0008              13 STEP     EQU     08H             ;STEP RATE
  0009              14 LOAD     EQU     09H             ;HEAD LOAD TIME
  4000              15 TCOUNT   EQU     4000H           ;TERMINAL COUNT AND DMA MODE  'OF
  0004              16 DMAMOD   EQU     04H             ;DMA MODE WORD
  0080              17 BUSY     EQU     80H             ;204 BUSY MASK
  0020              18 PARFUL   EQU     20H             ;204 PARAMETER REGISTER FULL  'AS
  0010              19 RESFUL   EQU     10H             ;204 RESULT BYTE FULL MASK
                    20          ;
                    21          ;               204 INITIALIZATION ROUTINE. RESETS 204 BOARD
                    22          ;               AND PERFORMS ALL OF THE NECESSARY INITIALIZATION
                    23          ;               OF THE 8257 AND 8271.
                    24          ;
                    25          CSEG
                    26          PUBLIC  INIT24          ;ENTRY POINT
                    27          PUBLIC  WAITC           ;WILL BE USED EXTERNALLY
                    28          PUBLIC  WAITP           ;
                    29 INIT24:
  0000 F3           30          DI                      ;DISABLE
  0001 3E0E         31          MVI     A,0EH           ;ENABLE 5.5 INTERRUPT
  0003 30           32          SIM                     ;
  0004 D3E4         33          OUT     MMSET           ;SET MASTER MODE FLIP FLOP
  0006 D38F         34          OUT     BASE+15         ;RESET INTERFACE
  0008 3E01         35          MVI     A,1             ;RESET 204
  000A D382         36          OUT     BASE+2          ;
  000C AF           37          XRA     A               ;
  000D D382         38          OUT     BASE+2          ;
  000F CD9900    C  39          CALL    WAITC           ;WAIT TILL COMMAND WRITE VAL?
  0012 3E35         40          MVI     A,SPECFY        ;OUTPUT "SPECIFY" COMMAND
  0014 D380         41          OUT     BASE+0          ;
  0016 CDA100    C  42          CALL    WAITP           ;WAIT TILL PARAMETER WRITE V? IC
  0019 3E0D         43          MVI     A,CHARS         ;SPECIFYING DRIVE CHARACTERISTIC
  001B D381         44          OUT     BASE+1          ;
  001D CDA100    C  45          CALL    WAITP           ;
  0020 3E08         46          MVI     A,STEP          ;OUTPUT STEP RATE
  0022 D381         47          OUT     BASE+1          ;
  0024 CDA100    C  48          CALL    WAITP           ;
  0027 3E08         49          MVI     A,SETTLE        ;OUTPUT HEAD SETTLE TIME
  0029 D381         50          OUT     BASE+1          ;
  002B CDA100    C  51          CALL    WAITP           ;
  002E 3E09         52          MVI     A,LOAD          ;OUTPUT HEAD LOAD TIME
  0030 D381         53          OUT     BASE+1          ;
  0032 CD9900    C  54          CALL    WAITC           ;
```

```
0035 3E35          55        MVI    A,SPECFY          ;SPECIFY BAD TRACKS
0037 D380          56        OUT    BASE+0            ;
0039 CDA100   C    57        CALL   WAITP             ;
003C 3E10          58        MVI    A,BADTR1          ;BAD TRACKS FOR SURFACE 1
003E D381          59        OUT    BASE+1            ;
0040 CDA100   C    59        CALL   WAITP             ;
0043 3EFF          61        MVI    A,NOBAD           ;FIRST TRACK
0045 D381          62        OUT    BASE+1            ;
0047 CDA100   C    63        CALL   WAITP             ;
004A 3EFF          64        MVI    A,NOBAD           ;SECOND BAD TRACK
004C D381          65        OUT    BASE+1            ;
004E CDA100   C    66        CALL   WAITP             ;
0051 3EFF          67        MVI    A,CTADDR          ;CURRENT TRACK ADDRESS (NOT F OW
0053 D381          68        OUT    BASE+1            ;
0055 CD9900   C    69        CALL   WAITC             ;
0058 3E35          70        MVI    A,SPECFY          ;
005A D380          71        OUT    BASE+0            ;
005C CDA100   C    72        CALL   WAITP             ;
005F 3E18          73        MVI    A,BADTR2          ;SURFACE 2
0061 D381          74        OUT    BASE+1            ;
0063 CDA100   C    75        CALL   WAITP             ;
0066 3EFF          76        MVI    A,NOBAD           ;FIRST TRACK
0068 D381          77        OUT    BASE+1            ;
006A CDA100   C    78        CALL   WAITP             ;
006D 3EFF          79        MVI    A,NOBAD           ;SECOND TRACK
006F D381          80        OUT    BASE+1            ;
0071 CDA100   C    81        CALL   WAITP             ;
0074 3EFF          82        MVI    A,CTADDR          ;CURRENT TRACK ADDRESS (NOT F OW
0076 D381          83        OUT    BASE+1            ;
0078 CD9900   C    84        CALL   WAITC             ;
007B 3E69          85        MVI    A,SEEK            ;SEEK TO TRACK 0
007D D380          86        OUT    BASE+0            ;
007F CDA100   C    87        CALL   WAITP             ;
0082 3E00          88        MVI    A,0               ;
0084 D381          89        OUT    BASE+1            ;
0086 D3E5          90        OUT    MMRSET            ;GO TO SLEEP WHILE 204 DOES IT
0088 FB            91        EI                       ;ENABLE INTERRUPTS
0089 76            92        HLT                      ;SLEEP
008A F3            93        DI                       ;DISABLE
008B 3E04          94        MVI    A,DMAMOD          ;SET DMA MODE
008D D388          95        OUT    BASE+8            ;
008F 3E00          96        MVI    A,LOW(TCOUNT)     ;SET CONTROL REGISTER
0091 D385          97        OUT    BASE+5            ;
0093 3E40          98        MVI    A,HIGH(TCOUNT)    ;
0095 D385          99        OUT    BASE+5            ;
0097 FB           100        EI                       ;
0098 C9           101        RET                      ;RETURN
                  102        ;
                  103        ;      WAITC AND WAITP ROUTINES
                  104        ;
0099 DB80         105 WAITC: IN     BASE+0            ;GET STATUS BYTE
009B E680         106        ANI    BUSY              ;BUSY?
009D C29900   C   107        JNZ    WAITC             ;YES,LOOP
00A0 C9           108        RET                      ;NO,RETURN
                  109        ;
00A1 DB80         110 WAITP: IN     BASE+0            ;GET STATUS REGISTER
00A3 E620         111        ANI    PARFUL            ;PARAMETER BUFFER FULL?
00A5 C2A100   C   112        JNZ    WAITP             ;YES,LOOP
00A8 C9           113        RET                      ;NO,RETURN
                  114        END
```

```
PUBLIC SYMBOLS
INIT24 C 0000    WAITC  C 0099    WAITP  C 00A1
```

EXTERNAL SYMBOLS

USER SYMBOLS
```
BADTR1 A 0010    BADTR2 A 0018    BASE   A 0080    BUSY   A 0080    CHARS  A 000D    CTA
INIT24 C 0000    LOAD   A 0009    MMRSET A 00E5    MMSET  A 00E4    NOBAD  A 00FF      AF
SEEK   A 0069    SETTLE A 0008    SPECFY A 0035    STEP   A 0008    TCOUNT A 4000    WAI
```

APPENDIX C

GRAPH 1
MASTER CPU FREE TIME
VS BAUD RATE
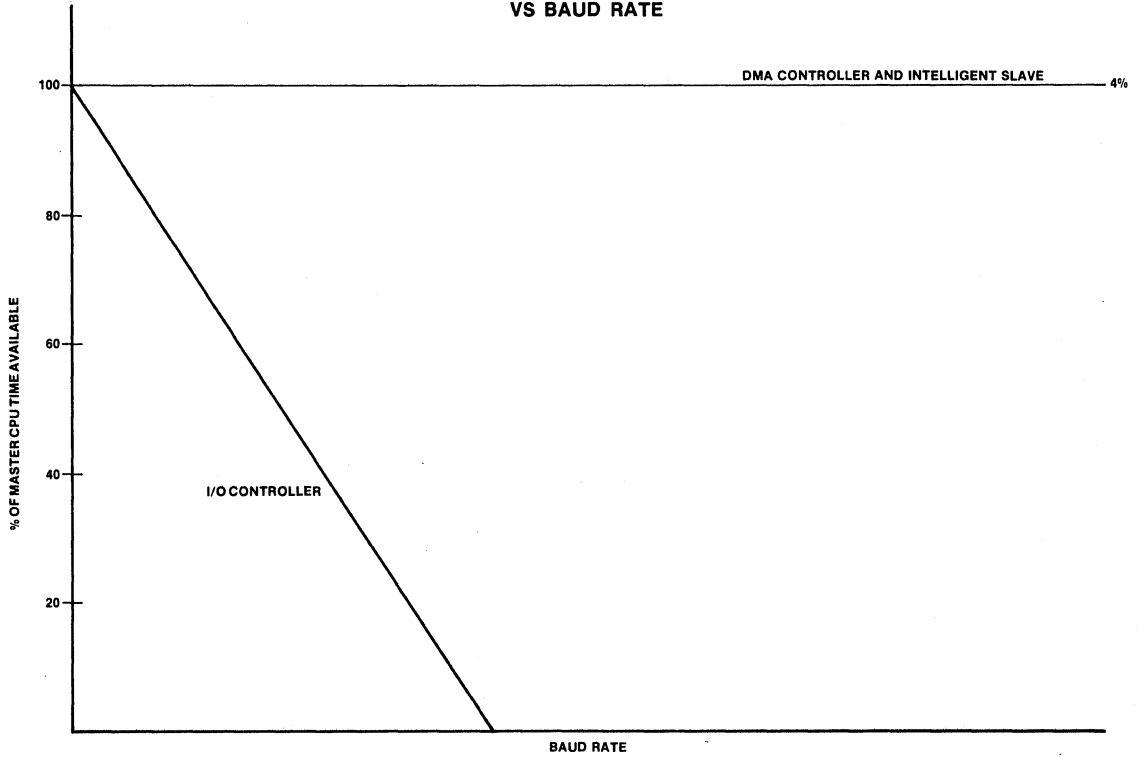
GRAPH 2
BUS FREE TIME
VS BAUD RATE
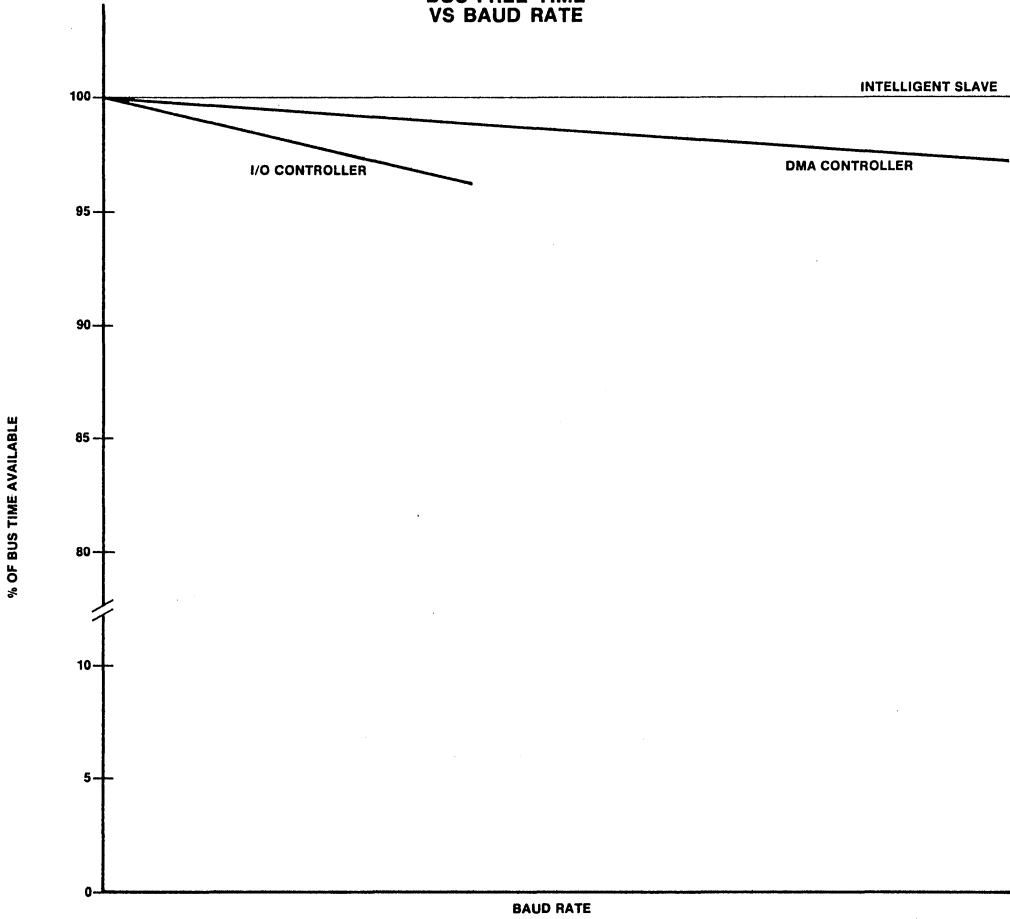
**GRAPH 3
SLAVE CPU FREE TIME
VS BAUD RATE**
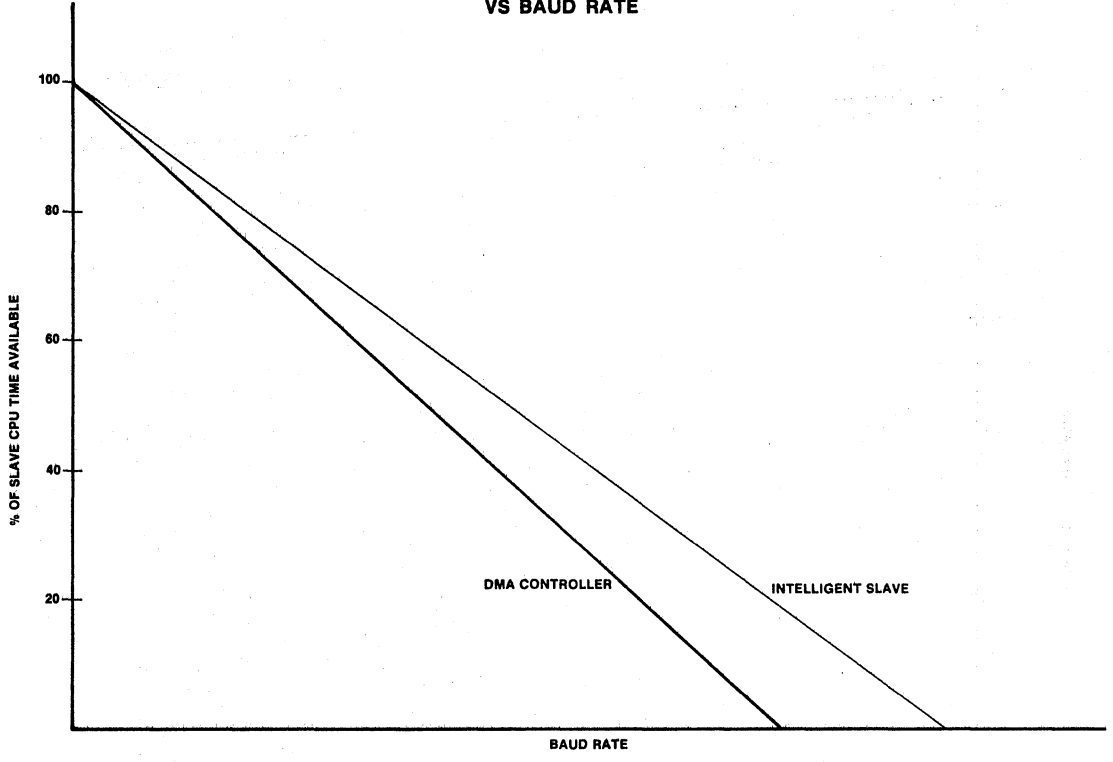
GRAPH 4
COMMUNICATIONS PROCESSOR FREE TIME
VS BUS TRAFFIC
@ 9600 BAUD FULL DUPLEX
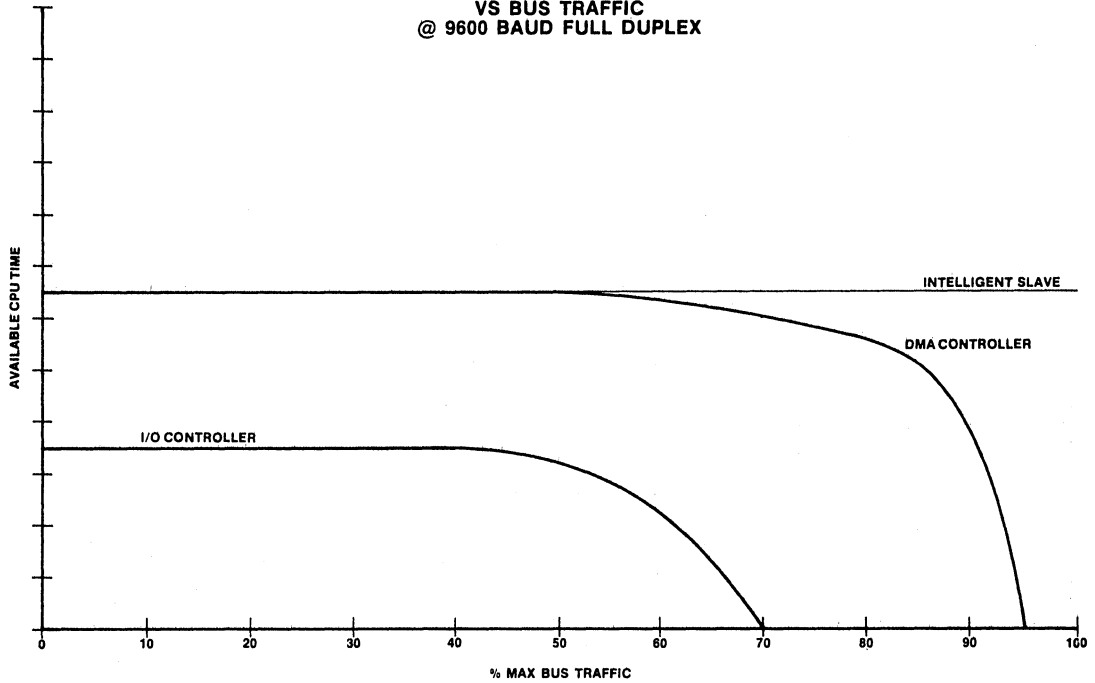
**GRAPH 5
MAXIMUM BAUD RATE
VS BUS TRAFFIC**
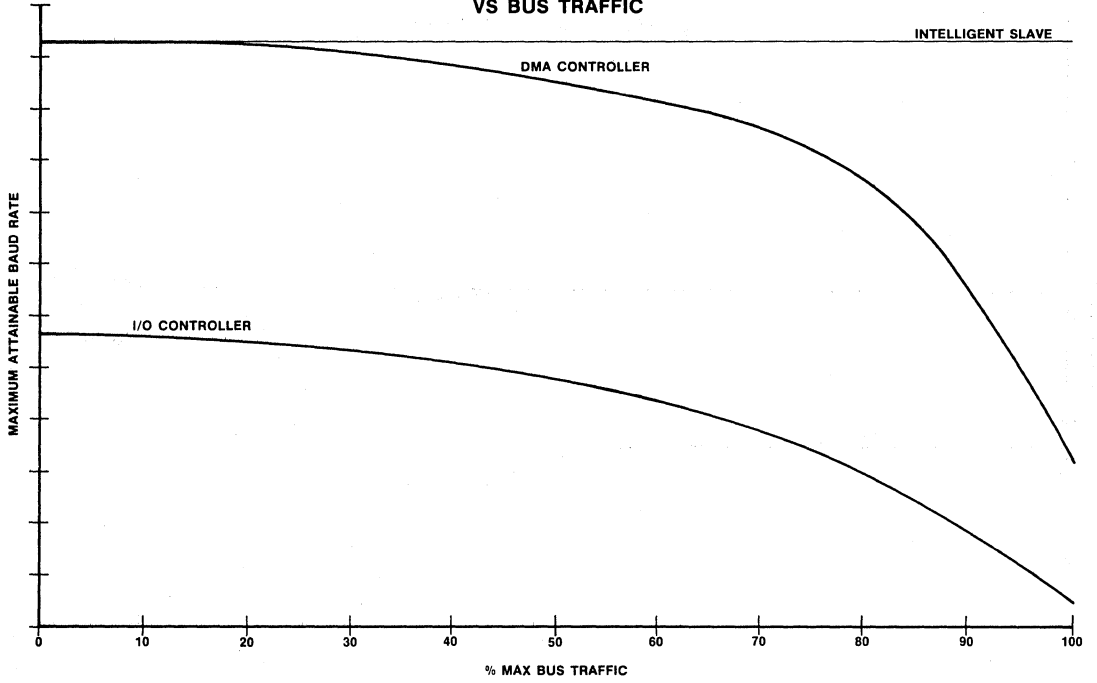
APPENDIX D

```
              $title('slave mainline routine')
     1        main$line:
                  DO;

              /*
                  Mainline routine. Sets up stack$ptr, calls s$init to init-
                  ialize queues,initializes some of the hardware, sets up the
                  initial flag interrupt handlers, and then halts with interru
          -   pts
                  enabled allowing the rest of the system to operate totally
                  in interrupt mode.
              */

              $nolist

    13    1   initial$handler:    PROCEDURE EXTERNAL;
    14    2   END initial$handler;


    15    1       DECLARE
                      command$word     LITERALLY    '41h',
                      port$a$8155 LITERALLY    '0e9h',
                      command$8155     LITERALLY    '0e8h',
                      mask$8259    LITERALLY    '0e7h',
                      icw1$8259    LITERALLY    '0e6h',
                      icw2$8259    LITERALLY    '0e7h',
                      ocw3$8259    LITERALLY    '0e6h',
                      read$isr     LITERALLY    '0bh',
                      mask$word BYTE PUBLIC,
                      port$a$value BYTE PUBLIC,
                      stat     BYTE,
                      i BYTE;

    16    1       output(icw1$8259)=0f6h;
    17    1       output(icw2$8259)=0fh;
    18    1       output(mask$8259),mask$word=0ffh;

    19    1       CALL s$init;

              /* set up 8259 for ISR reads */

    20    1       output(ocw3$8259)=read$isr;

    21    1       output(command$8155)=command$word;
    22    1       output(port$a$8155),port$a$value=0c0h;
    23    1       DO i=0 TO 7;
    24    2           stat=set$handler(i,.initial$handler);
```

```
25    2            END;

26    1            DO WHILE 1;
27    2                HALT;
28    2            END;

29    1       END main$line;
```

MODULE INFORMATION:

```
    CODE AREA SIZE      = 004DH      77D
    VARIABLE AREA SIZE  = 0004H       4D
    MAXIMUM STACK SIZE  = 0002H       2D
    72 LINES READ
    0 PROGRAM ERROR(S)
```

APPENDIX D (Continued)

ISIS-II PL/M-8Ø V3.1 COMPILATION OF MODULE INITIALHANDLER
OBJECT MODULE PLACED IN :Fl:FINTRT.OBJ
COMPILER INVOKED BY: PLM80 :Fl:FINTRT.PLM PRINT(:F5:FINTRT.LST) PAGEWIDTH(78)

```
            $title('slave application level signal handler')
    1       initial$handler:
                DO;

        /*
            Fields application level flag interrupts from the
            master. If the type=go$type the device attached to the queue
            specified is initialized with programming info sent into
            the queue by the master. If the type is data$available the
            specified transmitter is enabled unless a control$s pause
            is in effect.
        */

            $nolist

   32   1       DECLARE
                    no$pause     LITERALLY   '1',
                    go$type      LITERALLY   '1',
                    data$available  LITERALLY   '2',
                    enable$xmit LITERALLY    '1',
                    reset        LITERALLY   '40h',
                    timer$1$command$port LITERALLY  'Ødbh',
                    timer$2$command$port LITERALLY  'Ødfh',
                    mask$8259    LITERALLY   'Øe7h',
                    mask$word BYTE EXTERNAL,
                    mask (8) BYTE DATA(
                        Øfch,
                        Øfch,
                        Øf3h,
                        Øf3h,
                        Øcfh,
                        Øcfh,
                        Ø3fh,
                        Ø3fh),
                    transmitter$state (8) BYTE PUBLIC,
                    type     BYTE,
                    token    BYTE,
                    i    BYTE,
                    prog$info (5) BYTE,
                    actual ADDRESS,
                    usart$command$port (8) BYTE EXTERNAL,
                    usart$state (8) BYTE PUBLIC,
                    length$pointer (8) ADDRESS PUBLIC,
                    pointer (8) ADDRESS PUBLIC,
                    char$count (8) BYTE PUBLIC,
                    timer$load$port (8) BYTE DATA(
                        Ød8h,
```

```
                               0d8h,
                               0d9h,
                               0d9h,
                               0dah,
                               0dah,
                               0dch,
                               0dch);

33    1        initial$handler:     PROCEDURE (code) PUBLIC;

34    2           DECLARE code BYTE;

35    2           token=code AND 0fh;
36    2           type=shr(code,4);

37    2           IF type=go$type THEN
38    2           DO;
39    3               transmitter$state(token)=no$pause;
          /* reset usart */

40    3               DO i=0 TO 3;
41    4                   CALL varout(usart$command$port(token),0);
42    4               END;

43    3               CALL varout(usart$command$port(token),reset);

44    3               actual=get$line(token,.prog$info,5);

          /* program the devices */

45    3               CALL varout(usart$command$port(token),prog$info(0));
46    3               CALL varout(usart$command$port(token),usart$state(token)
      -    :=prog$info(1));
47    3               IF token < 7 THEN
48    3                   CALL varout(timer$1$command$port,prog$info(2));
                     ELSE
49    3                   CALL varout(timer$2$command$port,prog$info(2));
50    3               CALL varout(timer$load$port(token),prog$info(3));
51    3               CALL varout(timer$load$port(token),prog$info(4));

          /* open up the four input queues for data input */

52    3               length$pointer(token-1)=new$line(token-1);
53    3               pointer(token-1)=next$space(token-1);
54    3               char$count(token-1)=0;
55    3               output(mask$8259),mask$word=mask$word AND mask(token);
56    3           END;

                 ELSE
57    2           IF (type=data$available) AND (transmitter$state(token)=no$pa
      -    use) THEN
58    2           DO;
59    3               usart$state(token)=usart$state(token) OR enable$xmit;
60    3               CALL varout(usart$command$port(token),usart$state(token)
      -    );
61    3           END;
```

```
            RETURN;
63   2      END;

64   1      END initial$handler;
```

MODULE INFORMATION:

```
    CODE AREA SIZE     = 0182H    386D
    VARIABLE AREA SIZE = 0043H     67D
    MAXIMUM STACK SIZE = 0004H      4D
    154 LINES READ
    0 PROGRAM ERROR(S)
```

ISIS-II PL/M-80 V3.1 COMPILATION OF MODULE INPUTHANDLER
OBJECT MODULE PLACED IN :Fl:INHDLR.OBJ
COMPILER INVOKED BY: PLM80 :Fl:INHDLR.PLM PRINT(:F5:INHDLR.LST) PAGEWIDTH(78)


```
              $nointvector title('slave terminal input handler')
    1         input$handler:
                  DO;

          /*
              544 resident interrupt service routine. After receiver
              ready interrupt the 8259 In Service Register(ISR) is
              read to determine which device is requesting service.
              The character is read in and placed in the appropriate
              queue. A check is made for break characters and appropriate
              action is taken if any are found. When an endline character
              is encountered the length byte is filled in ( it was left
              vacant when the line was started) and the xmit primitive is
              called to update the global queue pointer to permit access
              to the line. At this time the master is signalled to signify
              that a new line is available for processing.
          */

              $nolist

   34   1      DECLARE
                      control$x    LITERALLY    '18H',
                      control$s    LITERALLY    '13H',
                      control$q    LITERALLY    '11H',
                      rubout       LITERALLY    '7FH',
                      escape       LITERALLY    '1BH',
                      CR       LITERALLY    '0DH',
                      LF       LITERALLY    '0AH',
                      BS       LITERALLY    '08H',
                      SP       LITERALLY    '20H',
                      bell         LITERALLY    '07H',
                      ptr      LITERALLY    'pointer(token)',
                      length$ptr   LITERALLY    'length$pointer(token)',
                      count        LITERALLY    'char$count(token)',
                      disable$xmit    LITERALLY    '0FEH',
                      enable$xmit LITERALLY    '01H',
                      no$pause         LITERALLY    '1',
                      pause    LITERALLY    '0',
                      line$available  LITERALLY    '1',
                      ocw2$8259    LITERALLY    '0E6H',
                      ocw3$8259    LITERALLY    '0E6H',
                      EOI      LITERALLY    '20H';

   35   1      DECLARE
                      value$ptr    ADDRESS,
                      value BASED value$ptr BYTE,
                      line$length BASED value$ptr BYTE,
                      dummy    ADDRESS,
                      ISR BYTE,
                      token    BYTE,
```

```
                      stat    BYTE,
                      new$ptr ADDRESS;

36    1          DECLARE
                      pointer (8) ADDRESS EXTERNAL,
                      length$pointer (8) ADDRESS EXTERNAL,
                      char$count (8) BYTE EXTERNAL,
                      usart$state (8) BYTE EXTERNAL,
                      usart$command$port (8) BYTE EXTERNAL,
                      usart$data$port (8) BYTE EXTERNAL,
                      transmitter$state (8) BYTE EXTERNAL;

37    1      index:  PROCEDURE (value) BYTE;
38    2          DECLARE value BYTE;

39    2          IF value=control$x THEN RETURN 0;
41    2          IF value=rubout THEN RETURN 1;
43    2          IF value=control$s THEN RETURN 2;
45    2          IF value=control$q THEN RETURN 3;
47    2          IF value=escape THEN RETURN 4;
49    2          IF value=CR THEN RETURN 5;
51    2          RETURN 6;
52    2      END;

53    1      echo:   PROCEDURE(token,buf$ptr,num$char) ADDRESS;

54    2          DECLARE (buf$ptr,num$char,actual) ADDRESS,
                      token BYTE;

55    2          actual=send$line(token,buf$ptr,num$char);
56    2          usart$state(token)=usart$state(token) OR enable$xmit;
57    2          CALL varout(usart$command$port(token),usart$state(token));
58    2          RETURN actual;
59    2          END;

60    1      delete$line:    PROCEDURE;

61    2          length$ptr=new$line(token);
62    2          ptr=next$space(token);
63    2          count=0;
64    2          dummy=echo(token+1,.('#',CR,LF),3);
65    2          RETURN;
66    2      END;

67    1      end$line:    PROCEDURE;

68    2          value$ptr=length$ptr;
69    2          line$length=count;
70    2          ptr=next$space(token);
71    2          stat=xmit(token);
72    2          length$ptr=new$line(token);
73    2          ptr=next$space(token);
74    2          count=0;
75    2          stat=set$s$interrupt(token,line$available);
76    2          RETURN;
77    2      END;
```

```
78    1        in$hdlr:     PROCEDURE INTERRUPT Ø PUBLIC;
79    2             ISR=input(ocw3$8259);

80    2             token=6;

81    2        again:
                   ISR=shl(ISR,2);

82    2             IF NOT carry THEN
83    2                 DO;
84    3                 IF token=Ø THEN RETURN; /* no bits set */
                        ELSE
86    3                     DO;
87    4                         token=token-2;
88    4                         GOTO again;
89    4                     END;

90    3                 END;
91    2             value$ptr=ptr;
92    2             value=varinp(usart$data$port(token)) AND Ø7fh;

93    2             DO CASE index(value);

                        /* case Ø; control$x; delete line */

94    3                 DO;
95    4                     CALL delete$line;
96    4                 END;

                        /* case 1; rubout; delete char */

97    3                 DO;
98    4                     new$ptr=back$space(token);
99    4                     IF new$ptr=length$ptr THEN
100   4                         dummy=echo(token+1,.(bell),1);
                            ELSE
101   4                         DO;
102   5                             dummy=echo(token+1,.(BS,SP,BS),3);
103   5                             ptr=new$ptr;
104   5                             count=count-1;
105   5                         END;
106   4                 END;

                        /* case 2; control$s; pause output */

107   3                 DO;
108   4                     usart$state(token+1)=usart$state(token+1) AND disabl
      -    e$xmit;
109   4                     CALL varout(usart$command$port(token+1),usart$state(
      -    token+1));
110   4                     transmitter$state(token+1)=pause;
111   4                 END;

                        /* case 3; control$q; resume output */
112   3                 DO;
113   4                     usart$state(token+1)=usart$state(token+1) OR enable$
```

```
               -    xmit;
114   4                      CALL varout(usart$command$port(token+1),usart$state(
               -    token+1));
115   4                      transmitter$state(token+1)=no$pause;
116   4                  END;

                        /* case 4; escape; terminate line */
117   3                  DO;
118   4                      dummy=echo(token+1,.('#',CR,LF),3);
119   4                      value=CR;
120   4                      count=count+1;
121   4                      CALL end$line;
122   4                  END;

                        /* case 5; carriage return; terminate line */
123   3                  DO;
124   4                      dummy=echo(token+1,.(CR,LF),2);
125   4                      count=count+1;
126   4                      ptr=next$space(token);
127   4                      value$ptr=ptr;
128   4                      value=LF;
129   4                      count=count+1;
130   4                      CALL end$line;
131   4                  END;

                        /* case 6; non-break character; stuff into queue */
132   3                  DO;
133   4                      dummy=echo(token+1,ptr,1);
134   4                      ptr=next$space(token);
135   4                      IF ptr=0 THEN CALL delete$line; /* full buffer */
137   4                      ELSE count=count+1;
138   4                  END;

139   3              END; /* of do case */
140   2              output(ocw3$8259)=EOI;

141   2              RETURN;

142   2          END;

143   1      END input$handler;
```

```
MODULE INFORMATION:

    CODE AREA SIZE      = 0398H      920D
    VARIABLE AREA SIZE  = 0011H       17D
    MAXIMUM STACK SIZE  = 0010H       16D
    255 LINES READ
    0 PROGRAM ERROR(S)
```

P /M-80 COMPILER   SLAVE CHARACTER OUTPUT HANDLER   PAGE   1

ISIS-II PL/M-80 V3.1 COMPILATION OF MODULE OUTPUTHANDLER
OBJECT MODULE PLACED IN :F1:OUTHLR.OBJ
COMPILER INVOKED BY: PLM80 :F1:OUTHLR.PLM PRINT(:F5:OUTHLR.LST) PAGEWIDTH(78)


```
          $nointvector title('slave character output handler')
  1       output$handler:
              DO;

          /*
              544 resident interrupt service routine. After transmitter
              ready interrupt, 8259 In Service Register(ISR) is read to
              determine which device is requesting service. A character
              is requested from the appropriate queue and, if available,
              is sent to the usart. If the queue is empty the transmitter
              is disabled pending a signal from the master when more
              characters are put into the queue.
          */

          $nolist

 11   1       DECLARE
                  ocw2$8259    LITERALLY    '0E6H',
                  ocw3$8259    LITERALLY    '0E6H',
                  disable$xmit    LITERALLY    '0FEH',
                  true         LITERALLY    '0FFH',
                  false        LITERALLY    '00H',
                  EOI     LITERALLY    '0A0H';

 12   1       DECLARE
                  ISR BYTE,
                  token    BYTE,
                  actual   ADDRESS,
                  value    BYTE;

 13   1       DECLARE
                  usart$state (8) BYTE EXTERNAL,
                  usart$command$port (8) BYTE PUBLIC DATA(
                      0D1H,
                      0D1H,
                      0D3H,
                      0D3H,
                      0D5H,
                      0D5H,
                      0D7H,
                      0D7H),
                  usart$data$port (8) BYTE PUBLIC DATA(
                      0D0H,
                      0D0H,
                      0D2H,
                      0D2H,
                      0D4H,
```

```
                         0D4H,
                         0D6H,
                         0D6H);
14    1        out$hlr:    PROCEDURE INTERRUPT 1 PUBLIC;

           /* get active level number and use it to determine queue$token *
      -    /

15    2            ISR=input(ocw3$8259);

16    2            token=7;

17    2        again:
                   ISR=shl(ISR,1);
18    2            IF NOT carry THEN
19    2                DO;
20    3                    IF token=1 THEN RETURN; /* no bits in ISR set */
                          ELSE
22    3                        DO;
23    4                            token=token-2;
24    4                            ISR=shl(ISR,1);
25    4                            GOTO again;
26    4                        END;
27    3                END;

28    2            actual=get$line(token,.value,1);
29    2            IF actual=0 THEN
30    2                DO; /* empty queue. Disable transmitter */
31    3                    usart$state(token)=usart$state(token) AND disabl
      -    e$xmit;
32    3                    CALL varout(usart$command$port(token),usart$stat
      -    e(token));
33    3                    END;
                      ELSE
34    2                CALL varout(usart$data$port(token),value);
35    2            output(ocw3$8259)=EOI;
36    2            RETURN;
37    2            END;
38    1        END output$handler;
```

MODULE INFORMATION:

```
    CODE AREA SIZE      = 00A4H    164D
    VARIABLE AREA SIZE  = 0005H      5D
    MAXIMUM STACK SIZE  = 000CH     12D
    102 LINES READ
    0 PROGRAM ERROR(S)
```

PL/M-80 COMPILER   RMX/80-544 INITIALIZATION TASK   PAGE   1

ISIS-II PL/M-80 V3.1 COMPILATION OF MODULE INIT544
OBJECT MODULE PLACED IN :Fl:INIT54.OBJ
COMPILER INVOKED BY: PLM80 :Fl:INIT54.PLM PRINT(:F5:INIT54.LST) PAGEWIDTH(78)


```
                    $title('rmx/80-544 initialization task')
      1             init$544:
                        DO;

                    /*
                        Task code for 544 driver initialization task. Info
                        from application supplied memory allocation block
                        is accessed to set up queues and transfer device programming
                        info to the slave board(s) and create the required
                        service handler tasks.
                    */

                    $nolist

     56   1         input$driver:   PROCEDURE EXTERNAL;
     57   2         END input$driver;

     58   1         output$driver:  PROCEDURE EXTERNAL;
     59   2         END output$driver;

     60   1         signal: PROCEDURE EXTERNAL;
     61   2         END signal;

     62   1             DECLARE
                            stack$size   LITERALLY   '256',
                            go$type      LITERALLY   '1';

     63   1             DECLARE
                            ptr      ADDRESS,
                            init$table BASED ptr STRUCTURE(
                                base$adr ADDRESS,
                                queue$token BYTE,
                                prog$info (5) BYTE),
                            i        BYTE,
                            overflow     ADDRESS,
                            queue$init$table (1) STRUCTURE(
                                base$adr ADDRESS,
                                queue$size (8) ADDRESS) EXTERNAL,
                            initialization$table (1) BYTE EXTERNAL,
                            stat         BYTE,
                            num$devices      BYTE EXTERNAL,
                            num$boards   BYTE EXTERNAL,
                            service$exchange$table (1) ADDRESS EXTERNAL,
                            signal$exchange$table (1) ADDRESS EXTERNAL,
                            service$exchanges (1) BYTE EXTERNAL,
                            signal$exchanges (1) BYTE EXTERNAL,
                            task$descriptors (1) BYTE EXTERNAL,
```

```
                    stacks (1) BYTE EXTERNAL,
                    info$block (1) STRUCTURE(
                        base$adr ADDRESS,
                        queue$token BYTE,
                        index BYTE) EXTERNAL,
                    rqactv      ADDRESS EXTERNAL;

64   1       DECLARE
                    rom$input$std static$task$descriptor DATA(
                        'input ',
                        .input$driver,
                        0, /* stack will be assigned individually */
                        stack$size,
                        200,
                        0, /* tba */
                        0), /* tba */
                    rom$output$std static$task$descriptor DATA(
                        'output',
                        .output$driver,
                        0,
                        stacksize,
                        201,
                        0,
                        0),
                    input$hdlr$std static$task$descriptor,
                    output$hdlr$std static$task$descriptor;

65   1    init$xch:   PROCEDURE (xch$ptr);
          /* initializes expanded interrupt exchanges */

66   2        DECLARE xch$ptr ADDRESS,
                    xch BASED xch$ptr int$exchange$descriptor;

67   2        xch.link=.xch.link;
68   2        xch.type=int$type;
69   2        xch.length=5;
70   2        RETURN;
71   2        END;

72   1    init$54: PROCEDURE PUBLIC;

73   2        DO i=0 TO num$boards-1;
74   3            CALL m$init(.queue$init$table(i));
75   3        END;

76   2        CALL move(size(rom$input$std),.rom$input$std,.input$hdlr$std
     -   );
77   2        CALL move(size(rom$output$std),.rom$output$std,.output$hdlr$
     -   std);
78   2        ptr=.initialization$table;

79   2        DO i=0 TO num$devices*2 BY 2;
          /* send pogramming info to slave */
80   3            overflow=send$line(init$table.base$adr,init$table.queue$
     -   token,.init$table.prog$info,5);
81   3            stat=set$m$interrupt(init$table.base$adr,init$table.queu
```

```
        -       e$token,go$type);

                /* create service and signal exchanges */

 82    3                CALL rqcxch(service$exchange$table(i):=.service$exchange
        -       s+10*i);
 83    3                CALL rqcxch(service$exchange$table(i+1):=.service$exchan
        -       ges+10*(i+1));
 84    3                CALL init$xch(.signal$exchanges+15*i);
 85    3                CALL init$xch(.signal$exchanges+15*(i+1));
 86    3                CALL rqcxch(signal$exchange$table(i):=.signal$exchanges+
        -       15*i);
 87    3                CALL rqcxch(signal$exchange$table(i+1):=.signal$exchange
        -       s+15*(i+1));

 88    3                info$block(i).base$adr,
                        info$block(i+1).base$adr=init$table.base$adr;
 89    3                info$block(i).queue$token=init$table.queue$token-1;
 90    3                info$block(i+1).queue$token=init$table.queue$token;
 91    3                info$block(i).index=i;
 92    3                info$block(i+1).index=i+1;

 93    3                input$hdlr$std.sp=.stacks+stack$size*i;
 94    3                output$hdlr$std.sp=.stacks+stack$size*(i+1);
 95    3                input$hdlr$std.exchange$address=.info$block(i);
 96    3                output$hdlr$std.exchange$address=.info$block(i+1);
 97    3                input$hdlr$std.task$ptr=.task$descriptors+20*i;
 98    3                output$hdlr$std.task$ptr=.task$descriptors+20*(i+1);

 99    3                CALL rqctsk(.input$hdlr$std);
100    3                CALL rqctsk(.output$hdlr$std);
101    3                ptr=ptr+8;
102    3            END;

103    2            CALL rqsetv(.signal,2);
104    2            CALL rqelvl(2);
105    2            CALL rqsusp(rqactv);
106    2        END; /* of task */

107    1        END init$544;
```

MODULE INFORMATION:

```
    CODE AREA SIZE      = 02C3H      707D
    VARIABLE AREA SIZE  = 002AH       42D
    MAXIMUM STACK SIZE  = 0006H        6D
    285 LINES READ
    0 PROGRAM ERROR(S)
```

```
P /M-80 COMPILER     RMX/80-544 INITIALIZATION MODULE AND              PAGE    1


ISIS-II PL/M-80 V3.1 COMPILATION OF MODULE INITMODULE
OBJECT MODULE PLACED IN :F1:MAB.OBJ
COMPILER INVOKED BY:   PLM80 :F1:MAB.PLM PRINT(:F5:MAB.LST) PAGEWIDTH(78)



            $title('rmx/80-544 initialization module and memory allocation b
       -    lock')
  1         init$module:
                DO;

            /*
                Initialization tables created and allocation of memory for 5
       -    44
                handler done here.
            */

  2    1      DECLARE
                  number$of$devices     LITERALLY    '4',
                  baud$rate$count$1     LITERALLY    '32',
                  baud$rate$count$h     LITERALLY    '00',
                  usart$mode          LITERALLY    '4eh',
                  usart$cmd           LITERALLY    '27h',
                  ctr$0$mode          LITERALLY    '36h',
                  ctr$1$mode          LITERALLY    '76h',
                  ctr$2$mode          LITERALLY    '0b6h',
                  ctr$3$mode          LITERALLY    '36h',
                  num$devices BYTE PUBLIC DATA(number$of$devices-1),
                  num$boards BYTE PUBLIC DATA(1),
                  service$exchange$table (8) ADDRESS PUBLIC,
                  signal$exchange$table (8) ADDRESS PUBLIC,
                  signal$type (8) BYTE PUBLIC,
                  service$exchanges (80) BYTE PUBLIC,
                  signal$exchanges (120) BYTE PUBLIC,
                  task$descriptors (160) BYTE PUBLIC,
                  stacks (2048) BYTE PUBLIC,
                  info$block (32) BYTE PUBLIC,
                  queue$init$table (1) STRUCTURE(
                      base$adr ADDRESS,
                      queue$size (8) ADDRESS) PUBLIC DATA(
                          0e000h,
                          256,
                          1765,
                          256,
                          1765,
                          256,
                          1765,
                          256,
                          1765),
                  base$table (1) ADDRESS PUBLIC DATA(
                          0e000h),
                  initialization$table (number$of$devices) STRUCTURE(
                      base$adr      ADDRESS,
```

```
                        queue$token BYTE,
                        prog$info (5) BYTE) PUBLIC DATA(
                            0e000h,
                            1,
                            usart$mode,
                            usart$cmd,
                            ctr$0$mode,
                            baud$rate$count$l,
                            baud$rate$count$h,

                            0e000h,
                            3,
                            usart$mode,
                            usart$cmd,
                            ctr$1$mode,
                            baud$rate$count$l,
                            baud$rate$count$h,

                            0e000h,
                            5,
                            usart$mode,
                            usart$cmd,
                            ctr$2$mode,
                            baud$rate$count$l,
                            baud$rate$count$h,

                            0e000h,
                            7,
                            usart$mode,
                            usart$cmd,
                            ctr$3$mode,
                            baud$rate$count$l,
                            baud$rate$count$h);
   3    1        END init$module;


MODULE INFORMATION:

      CODE AREA SIZE      = 0036H      54D
      VARIABLE AREA SIZE  = 09B0H    2480D
      MAXIMUM STACK SIZE  = 0000H       0D
      79 LINES READ
      0 PROGRAM ERROR(S)
```

P /M-80 COMPILER   SLAVE->MASTER INTERRUPT HANDLER   PAGE   1

ISIS-II PL/M-80 V3.1 COMPILATION OF MODULE SIGNALHANDLER
OBJECT MODULE PLACED IN :Fl:SIGNAL.OBJ
COMPILER INVOKED BY: PLM80 :Fl:SIGNAL.PLM PRINT(:F5:SIGNAL.LST) PAGEWIDTH(78)


```
              $nointvector title('slave->master interrupt handler')
    1         signal$handler:
                  DO;

              /*
                  Fields all slave->master signals(interrupts) and calls rqisn
          -   d
                  with the proper signal exchange address.
              */

              $nolist

   26    1         DECLARE
                      i        BYTE,
                      ptr      ADDRESS,
                      (flag BASED ptr) BYTE,
                      num$boards  BYTE EXTERNAL,
                      num$devices BYTE EXTERNAL,
                      signal$type (1) BYTE EXTERNAL,
                      index       BYTE,
                      token       BYTE,
                      signal$exchange$table (1) ADDRESS EXTERNAL,
                      base$table (1) ADDRESS EXTERNAL;

   27    1     signal: PROCEDURE INTERRUPT 2 PUBLIC;

              /* poll slave boards and find one generating interrupt */

   28    2         i=0;

   29    2     next:
                  ptr=base$table(i)+1;
   30    2         IF flag=0 THEN
   31    2             DO;
   32    3                 i=i+1;
   33    3                 IF i > num$boards THEN RETURN; /* erroneous signal *
          -   /
   35    3                 ELSE GOTO next;
   36    3             END;

              /* get queue token and use it to index into signal exchange tabl
          -   e */

   37    2         token=(flag AND 0fh);
   38    2         index=4*i+token;

              /* if index is out of range don't attempt the isend */
```

```
39   2           IF index <= num$devices THEN
40   2           DO;
41   3               CALL rqisnd(signal$exchange$table(index));
42   3               signal$type(index)=shr(flag,4);
43   3           END;

                 ELSE
44   2               CALL rqendi;

         /* zero flag to acknowledge interrupt */

45   2           flag=0;
46   2           RETURN;
47   2           END;

48   1       END signal$handler;
```

```
MODULE INFORMATION:

    CODE AREA SIZE      = 008BH    139D
    VARIABLE AREA SIZE  = 0005H      5D
    MAXIMUM STACK SIZE  = 000AH     10D
    110 LINES READ
    0 PROGRAM ERROR(S)
```

P /M-80 COMPILER   RMX/80-544 INPUT SERVICE HANDLER   PAGE   1

ISIS-II PL/M-80 V3.1 COMPILATION OF MODULE INPUTDRIVER
OBJECT MODULE PLACED IN :Fl:INPUT.OBJ
COMPILER INVOKED BY: PLM80 :Fl:INPUT.PLM PRINT(:F5:INPUT.LST) PAGEWIDTH(78)


```
            $title('rmx/80-544 input service handler')
   1        input$driver:
                DO;

            /*
                Master resident task code. Monitors service exchange
                and fills input requests by retrieving characters from
                the proper queue(board$base and device info is passed
                via default exchange field). By definition the first byte
                of a line of input contains the length of that line.
                This figure is used to retrieve the exact number of characte
      -    rs
                available in a given line.
            */

            $nolist

  27   1        DECLARE
                    rqactv       ADDRESS EXTERNAL,
                    td BASED rqactv task$descriptor,
                    service$exchange$table (1) ADDRESS EXTERNAL,
                    signal$exchange$table (1) ADDRESS EXTERNAL;

  28   1    input$driver:   PROCEDURE REENTRANT PUBLIC;

  29   2        DECLARE
                    service$exchange     ADDRESS,
                    board$base  ADDRESS,
                    queue$token BYTE,
                    signal$exchange ADDRESS,
                    msg$ptr       ADDRESS,
                    msg BASED msg$ptr th$msg,
                    actual        ADDRESS,
                    dummy         ADDRESS,
                    info$block$ptr  ADDRESS,
                    info$block BASED info$block$ptr STRUCTURE(
                        base$adr      ADDRESS,
                        queue$token BYTE,
                        index         BYTE),
                    num$char    BYTE,
                    stat        BYTE;

            /* get info out of default field */

  30   2        info$block$ptr=td.exchange$address; /* default exchange fiel
      -    d */
  31   2        service$exchange=service$exchange$table(info$block.index);
```

```
32   2           board$base=info$block.base$adr;
33   2           queue$token=info$block.queue$token;
34   2           signal$exchange=signal$exchange$table(info$block.index);
35   2           DO forever;

         /* wait for request message */

36   3               msg$ptr=rqwait(service$exchange,0);

37   3       retry:
         /* try to get line count out of queue */
                 actual=get$line(board$base,queue$token,.num$char,1);

         /* if unsuccessful wait for signal and try again */

38   3               IF actual=0 THEN
39   3               DO;
40   4                   dummy=rqwait(signal$exchange,0);
41   4                   GOTO retry;
42   4               END;

         /* if all okay get line */

43   3               actual=get$line(board$base,queue$token,msg.buffer$adr,nu
     -     m$char);
44   3               msg.actual=actual;
45   3               msg.status=0;
46   3               CALL rqsend(msg.resp$ex,msg$ptr);
47   3           END; /* of do forever */

48   2       END; /* of task */

49   1       END input$driver;
```

MODULE INFORMATION:

```
    CODE AREA SIZE      = 012CH      300D
    VARIABLE AREA SIZE = 0000H        0D
    MAXIMUM STACK SIZE = 0017H       23D
    171 LINES READ
    0 PROGRAM ERROR(S)
```

P /M-80 COMPILER   RMX/80-544 OUTPUT SERVICE HANDLER   PAGE   1

```
              $title('rmx/80-544 output service handler')
  1           output$driver:
                  DO;

              /*
                  Master resident task code. Monitors service exchange and
                  fills output requests by stuffing characters into the approp
          -   riate
                  queue. If insufficient room is available the task waits
                  for 1 second and retries up to 100 times after which it
                  signals a time out error. If the transmission completes
                  successfully the slave is signalled to indicate that data is
          -
                  available.
              */

              $nolist

 39    1          DECLARE
                      data$available  LITERALLY    '2',
                      time$out      LITERALLY    '1';

 40    1          DECLARE
                      rqactv  ADDRESS EXTERNAL,
                      (td BASED rqactv) task$descriptor,
                      service$exchange$table (1) ADDRESS EXTERNAL,
                      signal$exchange$table (1) ADDRESS EXTERNAL;

 41    1      output$driver:  PROCEDURE REENTRANT PUBLIC;

 42    2          DECLARE
                      service$exchange     ADDRESS,
                      signal$exchange      ADDRESS,
                      base$adr          ADDRESS,
                      queue$token       BYTE,
                      msg$ptr           ADDRESS,
                      msg BASED msg$ptr    th$msg,
                      tries$left        BYTE,
                      overflow          ADDRESS,
                      dummy             ADDRESS,
                      stat              BYTE,
                      info$block$ptr        ADDRESS,
                      info$block BASED info$block$ptr STRUCTURE(
                          base$adr      ADDRESS,
                          queue$token BYTE,
                          index       BYTE);
```

```
            /* initialize */

43    2         info$block$ptr=td.exchange$address;
44    2         service$exchange=service$exchange$table(info$block.index);
45    2         signal$exchange=signal$exchange$table(info$block.index);
46    2         base$adr=info$block.base$adr;
47    2         queue$token=info$block.queue$token;

48    2         DO forever;

            /* wait for request message */

49    3             msg$ptr=rqwait(service$exchange,0);
50    3             tries$left=100;
51    3     retry:
                   overflow=send$line(base$adr,queue$token,msg.buffer$adr,m
      -     sg.count);
52    3             IF overflow <> 0 THEN
53    3             DO;
54    4                 dummy=rqwait(signal$exchange,20);
55    4                 tries$left=tries$left-1;
56    4                 IF tries$left > 0 THEN GOTO retry;
                       ELSE
58    4                     DO;
59    5                         msg.status=time$out;
60    5                         msg.actual=0;
61    5                         GOTO quit;
62    5                     END;
63    4             END;
64    3             msg.status=0;
65    3             stat=set$m$interrupt(base$adr,queue$token,data$available
      -     );
66    3             msg.actual=msg.count;
67    3     quit:
                   CALL rqsend(msg.resp$ex,msg$ptr);
68    3         END; /* of do forever */

69    2     END; /* of task */

70    1     END output$driver;
```

MODULE INFORMATION:

```
    CODE AREA SIZE     = 0159H     345D
    VARIABLE AREA SIZE = 0000H       0D
    MAXIMUM STACK SIZE = 0019H      25D
    198 LINES READ
    0 PROGRAM ERROR(S)
```

A M8Ø :Fl:CFG544.M8Ø PRINT(:F4:CFG544.LST) PAGEWIDTH(78) MACROFILE

ISIS-II 8Ø8Ø/8Ø85 MACRO ASSEMBLER, V3.Ø   CFG544   PAGE   1


```
  LOC  OBJ           LINE           SOURCE STATEMENT

                       1            NAME    CFG544
                       2            CSEG
                       3            PUBLIC  RQRATE
 ØØØØ Ø8ØØ             4  RQRATE: DW      8
                       5  $NOLIST
                     127  $LIST
                     128  $NOGEN
 ØØØØ                129  NTASK   SET     Ø
 ØØØØ                130  NEXCH   SET     Ø
                     131 ;
                     132 ;      BUILD THE INITIAL TASK TABLE
                     133 ;
                     134 ; --------\  THIS TASK IS NECESSARY FOR THE 544 HANDLE
                         R
                     135 ; --------/  IT CREATES EVERYTHING ELSE IT NEEDS.
                     136         STD     INIT54,2ØØ,1,Ø
                     191         STD     LINECH,64,13Ø,Ø
                     246
                     247 ;      ALLOCATE TASK DESCRIPTORS
                     248 ;
                     249         GENTD
                     253 ;
                     254 ;      BUILD INITIAL EXCHANGE TABLE
                     255 ;
                     256         XCHADR  RESPEX
                     264 ;
                     265 ;      BUILD CREATE TABLE
                     266 ;
                     267         CRTAB
                     274         END
```
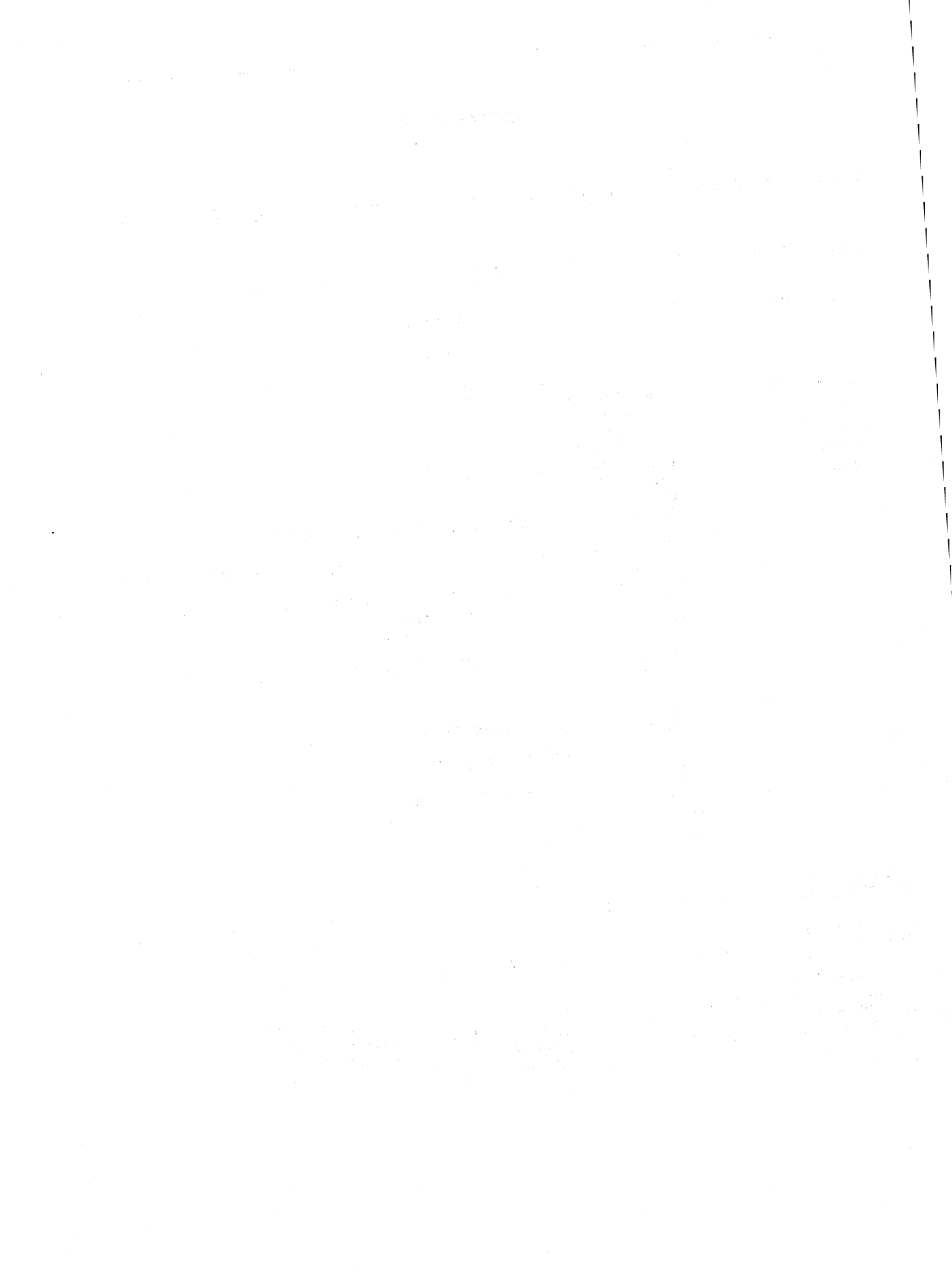
```
PUBLIC SYMBOLS
RQCRTB C ØØ26    RQRATE C ØØØØ

EXTERNAL SYMBOLS
INIT54 E ØØØØ    LINECH E ØØØØ    RESPEX E ØØØØ

USER SYMBOLS
CRTAB  + ØØØØ    GENTD  + ØØØØ    IET    C ØØ24    INIT54 E ØØØØ
INTXCH + ØØØØ    ITT    C ØØØ2    LINECH E ØØØØ    NEXCH  A ØØØ1
NTASK  A ØØØ2    PUBXCH + ØØØ7    RESPEX E ØØØØ    RQCRTB C ØØ26
RQRATE C ØØØØ    STD    + ØØØØ    TDBASE D Ø1Ø8    XCH    + ØØØ5
XCHADR + ØØØ2
```

intel®

JUNE 1977

Intel® SBC 80/10
Single Board Computer

John Haynes
Reliability Engineer
Microcomputer Division

Intel® SBC 80/10 Single Board Computer Reliability Report

Contents

INTRODUCTION

This report presents reliability test and supporting data for the Intel® SBC 80/10 Single Board Computer, a complete computer system on a single 6.75-by-12 inch printed circuit board. The CPU, system clock, read/write memory, non-volatile read-only-memory, I/O ports and drivers, serial communications interface, bus control logic and drivers all reside on the board.

A complete life testing program has been implemented to ascertain and quantify the reliability level of the Intel® SBC 80/10 Single Board Computer. This report presents a review of this program, including a discussion of the test configuration utilized, the system exerciser/diagnostic software used, the number of hours of testing performed, and the ambient test temperatures. High temperature testing was performed to accelerate product aging and to provide operational data on the reliability performance of the SBC 80/10 computer at elevated temperatures.

Data gathered from this life testing shows the SBC 80/10 to provide extremely reliable performance. MTBF for the SBC 80/10 Single Board Computer is shown to be 91,739 hours (90% confidence) in continuous +25°C environments (average "room" temperature) and 25,006 hours (90% confidence) in continuous +55°C high temperature environments (maximum operating temperature specified for SBC 80 products). Data is also

presented which shows that this life test data correlates well with actual field reliability data.

A full description of Intel's complete component testing and quality assurance and board-level assembly, testing, and quality assurance procedures is also included in this report. This will provide an understanding of the production, testing, and quality assurance procedures which all Intel® SBC 80 Single Board Computer products undergo prior to shipment. This report is presented as a basis for reliability performance estimates for Intel's complete SBC 80 line of Single Board Computers and supporting expansion boards.

SBC 80/10 COMPUTER DESCRIPTION

The SBC 80/10 Single Board Computer is contained on a 6.75-by-12 inch printed circuit board. As Figure 1 shows, the CPU, system clock, read/write memory, non-volatile read-only-memory, I/O ports and drivers, serial communications interface, bus control logic and drivers all reside on the board. Intel n-channel MOS LSI technology and bipolar TTL are the basis for the feasibility of such a full computer on a single PC board. Such a single board approach is inherently more reliable than traditional multiboard implementations through minimization of inter-board connectors.

The heart of the SBC 80/10 Single Board Computer is the Intel® 8080A CPU which contains six
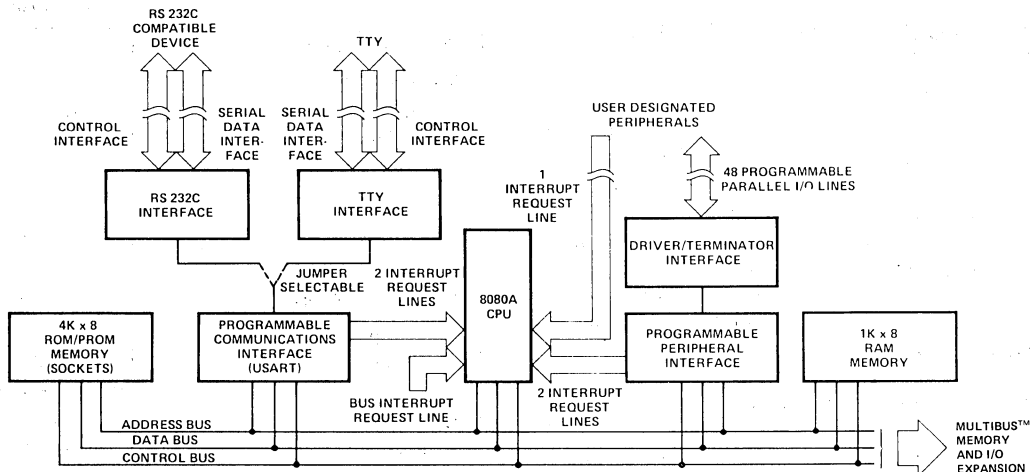
1. Interrupts originating from the Programmable Communications Interface and Programmable Peripheral Interface are jumper selectable.

Figure 1. SBC 80/10 Block Diagram

8-bit, general-purpose registers and an accumulator. Complete reliability data on the 8080A microprocessor is contained in Intel Reliability Report RR-10. 1024 bytes of read/write storage are provided by eight Intel low-power static RAMs. In addition to read/write memory, four sockets are provided to install four Intel 8708 1024 × 8 nonvolatile, erasable and reprogrammable read-only-memories. The Intel 8708 EPROM has been shown to be an extremely reliable Read-Only-Memory (see Intel Reliability Report RR-12 for complete details).

Two Intel 8255 Programmable Peripheral Interface devices supply 48 parallel I/O lines. These lines are routed to sockets on the board which are used to buffer lines programmed as outputs with TTL drivers or to terminate lines programmed as inputs with SBC 901 or SBC 902 resistor terminator packs.

For serial communication, an Intel 8251 Universal Synchronous/Asynchronous Receiver/Transmitter (USART) is contained on the SBC 80/10 computer. Additional circuitry provides jumper-selectable teletypewriter (20 mA current loop) or RS232C compatible interfaces for the USART.

For further information on detailed operation of the SBC 80/10 computer, consult the Intel SBC 80/10 Single Board Computer Hardware Reference Manual.

SINGLE BOARD COMPUTER QUALITY ASSURANCE

The reliability of a board level product like the Intel® SBC 80/10 Single Board Computer is a function of the quality of components used on the board, care in board design and fabrication, and the extent of testing performed on the product before shipment. An examination of each of these functions will provide an understanding of the Intel quality assurance program for microcomputer system products.

Component Quality Assurance

Standard Intel® component quality assurance processing and 100% screening are applied to all Intel components used on the SBC 80/10 computer before they are assembled on the board. Once a component device has been qualified as

reliable and transferred to systems level manufacturing, it is imperative that complete process controls are in effect to assure the continuation of high quality. Intel's component QA flow centers around a series of acceptance gates between process entities (see Figure 2) and detailed inspection within the processing areas at critical points. For example, in wafer processing, furnaces are routinely monitored for contamination through the use of capacitance voltage measurements on test chips. Also, electrical tests such as breakdown strength measurements are performed on test patterns on each wafer. Routine high magnification scanning electron microscope examinations at critical process steps also provide important process control feedback. A final QA acceptance is performed on all lots prior to shipment to assembly locations. Table I details the Intel component QA acceptance gates within the assembly flow and identifies the appropriate method employed per MIL-STD-883. During the test and finishing operation, QA maintains standards, test tape and calibration control of all production equipment. Final QA inspection is performed after the mark and pack operation.

Table I

COMPONENT ASSEMBLY FLOW

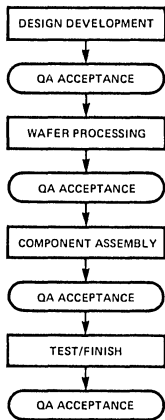| OPERATION	MIL-STD-883 METHOD
Piece Part Inspections	
Scribe and Break	
QA Acceptance	2010.1B
2nd Optical Inspection	2010.1B
Lead Bond	
QA Acceptance	2010.1B
3rd Optical Inspection	2010.1B
Seal	
Temperature Cycle	1010 (C)
QA Acceptance	1014 (B)
Fine Leak	1014 (A)
	(10^{-7} atm cc/sec)
QA Acceptance	1014 (C)
Gross Leak	1014 (C)
QA Acceptance	2009
Final Visual	2009

Figure 2. Component QA Flow Diagram

Board Assembly

Figure 3 shows the stringent cycle of inspection and testing each board undergoes before being shipped to the customer. After bare board manufacture, boards are inspected for raw board quality conformance (i.e., hole registration, drilling, laminate integrity, artwork registration, etc.). Selected samples are sectioned and microscopically analyzed for compliance to specification.
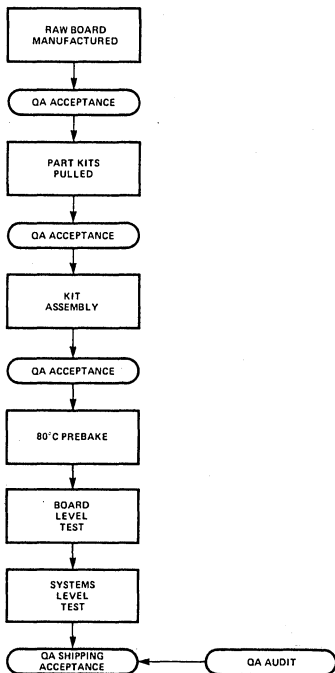
Figure 3. Board Manufacture and Test Flow/Screening

Components for an assembly "kit" are then pulled together and readied for the assembly operation. Visual inspection of each kit of components is then conducted. The kit is assembled onto the bare board and the assembled parts of the board are inspected prior to wave solder for proper location and after wave solder for soldering defects. Final assembly inspection insures the boards are ready for test.

Board Production Testing

Manufacturing testing, though not a part of reliability testing per se, is very effective in eliminating "infant mortality" failures. These types of failures, by their nature, occur in the early life of a product (see page 5). Each SBC 80/10 Single Board Computer is pre-baked at 80°C prior to any testing. Figure 4 shows some of the ovens utilized in this process. The boards are allowed to cool and then exhaustively tested on a Teradyne L125 Digital Diagnostic Tester. Use of a special "bed of nails" vacuum test fixture insures that all critical board test points may be accessed by the test system. Figure 5 shows the test fixture for the SBC 80/10 computer, and Figure 6 shows one of the Teradyne test systems used in Intel® microcomputer board-level testing. Failed components are replaced, and the boards are retested on the Teradyne test system.

Figure 4. Ovens Utilized for Board Pre-Bake

When the boards have passed this thorough testing, they are then "system tested" to guarantee Intel® MULTIBUS™ interface integri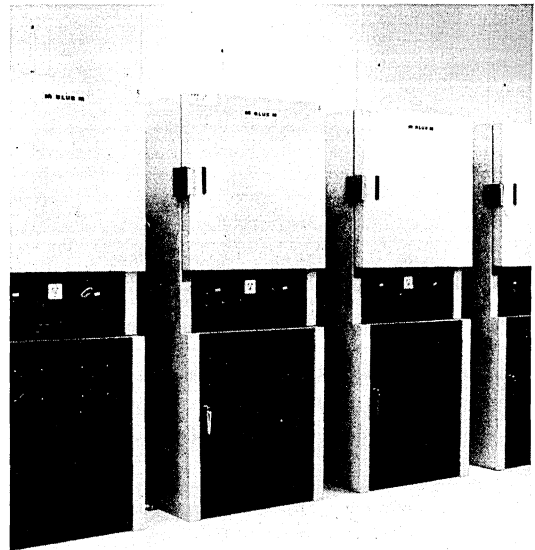ty. This consists of installing a set of Monitor and diagnostic program memory EPROMs into the four SBC 80/10 EPROM sockets, applying all necessary voltages, and monitoring the SBC 80/10 computer's performance with a teletypewriter. The board performs all of the functions as described later in the reliability test section. After test, the boards are inspected for possible test damage. Boards passing this final screen are then released by Intel Quality Assurance to Intel's standard product warehouse. Each product shipped is also subject to an audit. This audit consists of a complete review of documentation, test, functional performance workmanship, and handling.

THE SBC 80/10 RELIABILITY TEST

In order to establish reliability data for the SBC 80/10 Single Board Computer, an extensive life test program has been implemented. Twenty-one SBC 80/10 computers have been tested under various operating conditions. The various aspects of this program are discussed below.

Reliability Test

The test program used to exercise each SBC 80/10 computer during life testing was a modified version of the normal systems test, monitor, and diagnostic exerciser programs that each unit must pass prior to shipment. Figure 7 illustrates the hardware configuration of the test. Four Intel® 8708 EPROMs containing the monitor/diagnostic exerciser were installed in each SBC 80/10 computer to exercise virtually all of its functions.

Figure 5. SBC 80/10   Single Board Computer Test Fixture

Figure 6. A Teradyne Test System Utilized in SBC 80 Board Testing
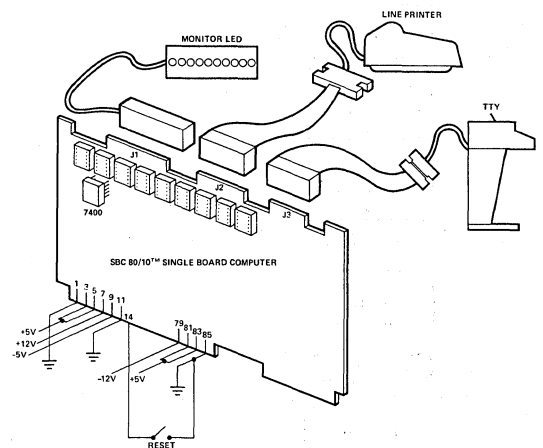
EQUIPMENT USED:
Power Supplies:
Power One   G5-35/OUP   S113
Lambda   LXS-D-5-0V

Test Equipment:
Fluke   800A
Fluke   2240A
Delta Design   6300/6400

Communication:
Teletype   Model 33
Centronics   306C

Figure 7. Life Test Configuration

The monitor program was a basic conditioning and communicative program. Among other things, it conditioned each SBC 80/10 computer to communicate with a teletypewriter, allowed the operator to display and/or change the contents of RAM memory locations and direct the program counter to any designated memory location to begin processing instructions. The monitor program occupied the first 1K of memory and started processing at location 0000H.

The diagnostic exerciser program, starting address 0400H, is arranged into two major sections: the CPU/memory Test and the I/O Test. The Appendix provides a flowchart of this reliability test program.

CPU/Memory Test

The test program first sets aside sections of RAM memory to be used as temporary storage needed in later processing. It then begins executing the instruction set of the 8080A CPU, starting with conditional jumps, adds, subtracts, and compare instructions. Six sequential conditional call and return instructions are executed with combinations of register manipulations. All registers are checked for proper incrementing, decrementing, and comparing. Different combinations of register compares and conditional jumps are done, followed by data move operations from register to register. A test of the and, or, and exclusive or instructions is then run. Each byte of a double register (16-byte) is incremented and decremented separately to see the effect on its companion register and on the Processor Status Word (flags, carry, parity, etc.). Exchange, register store, direct register add, decimal adjust, rotate left and rotate right instructions are then performed. Finally, a series of push and pop instructions combined with store directs and exchanges are executed to test as many application sequences as possible.

The memory is well exercised as a part of the CPU test. The memory test executes a complete series of store directs into and load directs from memory via double CPU registers. This is done to insure proper functioning of all memory locations.

Input/Output (I/O)

One of the Intel® 8255 Parallel I/O Interfaces is addressed and programmed to be a set of three output ports. Alternate bit patterns 10101010 and 01010101 (binary) are presented to each port and

LEDs are attached to allow monitoring. The second Intel® 8255 is programmed as a set of three output ports. One of these ports is used to drive a line printer in recording failures.

Serial I/O is tested periodically by manually resetting the board and using a teletypewriter communicating with the Monitor program. The automatic portion of the test is then restarted from this mode.

Error Handling

After the execution of selected compares and jumps, a routine is executed to record the occurrence of any errors on a line printer and to stop processing. If the failure was catastrophic, the board would be removed and trouble-shot to determine the failure mechanism.

Temperature

Twenty-one boards were tested at ambient temperatures of +25°C, +55°C, and +70°C. The +55°C and +70°C elevated temperatures were used to increase stress on the boards and stimulate failures. (Tables IV and V on page 9 present the number of unit-hours run at each temperature.)

BOARD RELIABILITY AND THE LIFE CURVE

There is a fundamental principle of reliability engineering which predicts that the failure rate of any group of products implemented from integrated circuits will follow a curve such as that shown in Figure 8.

This curve can be divided into three major categories of failures: Infant Mortality, Random, and Wear Out.
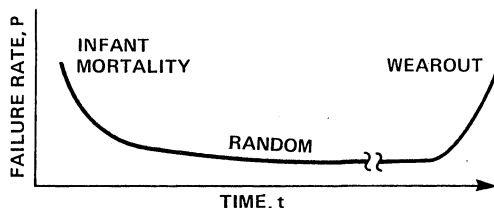

Figure 8. Reliability Life (Bathtub) Curve

Infant Mortality is that area of the curve where failures caused by manufacturing defects in both the components and/or the board are most apparent. Infant Mortality usually lasts from board manufacture to the low hundreds of hours of board operation.

Random failures occur during the useful portion of the product life. These failures are a function of temperature, circuit complexity, device loading, and other factors. The Random failure rate approaches a low constant value where it remains from hundreds to hundreds of thousands of operating hours (for printed circuit boards with integrated circuits). Since it is the parameter of greatest interest to the system designer, the failure rate for SBC 80/10 Single Board Computers through the Infant and Random failure regions was explored in reliability testing. During these tests, no evidence of wear out was experienced. Wear-out failures, as the name implies, occur when wear-out takes place, both physically and electrically, at the end of a device's useful life. Statistically, this will not happen until hundreds of years have elapsed for LSI-based products such as the SBC 80/10 Single Board Computer.

Reliability Mathematics

When information on the failure rate for a given LSI-based product is desired, that product generally must have its life "accelerated" due to the low failure rates involved. This can be done by subjecting the unit to high temperature, high voltage, or both. When high temperature is used, the thermal activation energy of failures can be used along with the failure rate at that temperature, to derive the effective failure rate at other temperatures. Such derivation is accomplished using the plot shown in Figure 9. This is an Arrhenius plot in which average time to failure (inverse of failure rate) is plotted against the reciprocal of temperature. The slope of the curve is related to the thermal activation energy in electron-volts. From these plots the effective failure rate at a given temperature may be derived from failure rate data determined at a different temperature. The two failure rates are related as follows:

$$ET_{T_1} = MF\ (AT_{T_2})$$

where:

ET_{T_1} = Effective Time (hours) at T_1 (Ambient Temperature)

MF = Multiplication Factor

AT_{T_2} = T_2 (Test Temperature)

The Multiplication Factor (MF) is found from the expression:

$$MF = \exp \left\{ + \frac{E}{k} \left(\frac{1}{T_1} - \frac{1}{T_2} \right) \right\}$$

where:

MF = Multiplication Factor

E = Thermal Activation Energy (eV)

k = Boltzmann's Constant
$(3.63 \times 10^{-5}\ eV/°K)$

T_1 = Ambient Temperature

T_2 = Test Temperature
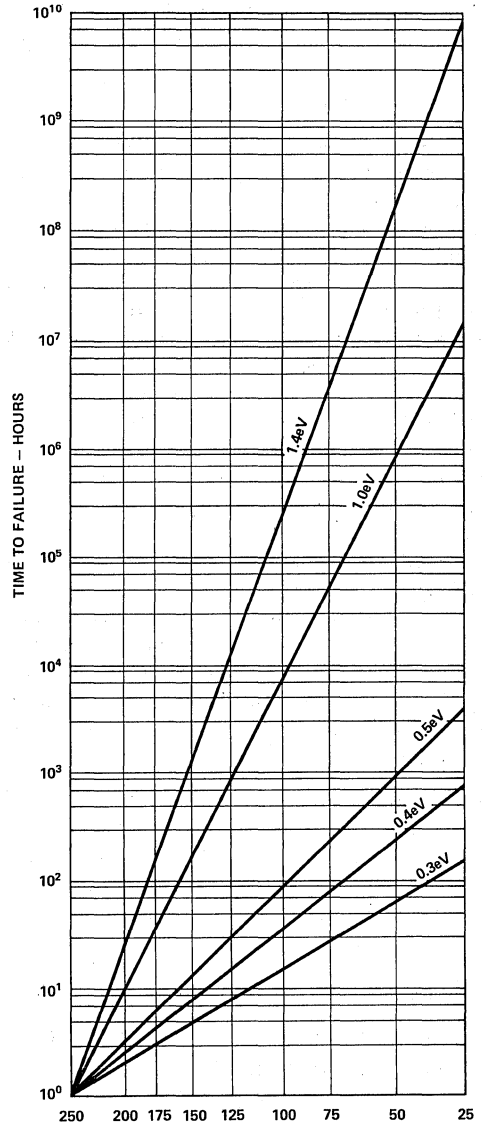
Figure 9. Arrhenius Plot

The thermal activation energy used in the calculation is determined by the failure mechanism. Table II lists the most common failure mechanisms for MOS LSI devices. As can be seen from Table II, activation energies for n-channel MOS devices range from 0.3 eV to 1.4 eV. Bipolar devices have been shown to exhibit activation energy performance at about 0.5 eV. Table III lists the most common failure mechanisms for printed wiring assemblies.

A mixture of n-channel MOS LSI, bipolar SSI and MSI, and discrete components are used on the SBC 80/10 Single Board Computer. The activation energy of the one failure during SBC 80/10 life testing could not accurately be determined. The failure occured in an Intel® 8224 clock generator at 460 hours into the test at an ambient temperature of +25°C. This clock generator is sensitive to a +12V/–12V reversal of supply voltages which happened once during the test. This may explain the one failure shown in Table IV.

The Intel® 8080 life test program produced results indicating 0.5 eV as an appropriate activation energy for failure rate analysis (see Intel Reliability Report RR-10). An activation energy of 0.5 eV was used here as a conservative basis for failure rate conversion.

Confidence Levels

When calculating a failure rate, the simplest form of calculation is:

$$\text{Failure Rate} = \frac{\text{Number of Failures}}{(\text{No. of Devices}) \times (\text{No. of Hours Tested})}$$

The only problem in this equation is that the failure rate thus calculated does not reflect a very high degree of confidence. In fact, the confidence level is only 50% for this equation. Since failures occurring beyond "Infant Mortality" and before product "wear out" are random in nature, the average failure rate for products operating in this region will

Table II

COMPONENT FAILURE MODES

FAILURE MODE	TYPE	ACTIVATION ENERGY (E_{act})	DETECTION
Slow Trapping	Wear Out	1.0 eV	High Temperature Bias
Contamination	Wear Out/Infant	1.4 eV	High Temperature Bias
Surface Charge	Wear Out	0.5–1.0 eV	High Temperature Bias
Polarization	Wear Out	1.0 eV	High Temperature Bias
Electromigration	Wear Out	1.0 eV	High Temperature Operating Life
Microcracks	Random	–	Temperature Cycling
Contacts	Wear Out	–	High Temperature Operating Life
Oxide Defects	Infant/Random	0.3 eV	High Voltage Operating Life and Cell Stress

Table III

PRINTED CIRCUIT FAILURE MECHANISMS

FAILURE MODE	TYPE	DETECTION	PREVENTATIVE MEASURES
Fractured Wirewrap	Random	Visual & Functional	Proper Tool Handling
Gold Finger Contamination	Random	Visual & Functional	Cleanliness, Gold Thickness

follow a Normal probability distribution. Such a distribution is shown in Figure 10. In Figure 10, the failure rate curve shows failure rate on the abscissa (X axis) and the failure frequency on the ordinate (y axis). The 50% confidence level is synonymous with the mean point of the curve and simply means that, in the population of product units, 50% of the units will have a failure rate less than or equal to the observed failure rate. The 90% confidence level is the 90th percentile of the distribution and means that 90% of the parts will have a failure rate less than the quoted number.

Of course, the failure rate at the 90th percentile is much higher than at the 50th percentile and determination of the failure rate at a 90% confidence level of greatest interest. Since the failure rate for random failures follows a Normal probability distribution, the Chi-squared probability function provides a direct tool for derivation of failure rates at various confidence levels. Using the Chi-squared function, the failure rate can be calculated at any confidence level by:

$$\text{Failure Rate} = \frac{\chi^2 (1 - CL, \ 2r + 2)}{2nt}$$

where:

χ^2 = Chi-square function

CL = Confidence level expressed as a decimal

r = Number of rejected boards

n = Number of boards tested

t = Total test time

The values of χ^2 may be found in most texts on statistics where the $2r + 2$ term is treated as the degree of freedom (DF). Given these tools for analysis, let us fully examine the results of the life test performed.

LIFE TEST RESULTS

The results of the SBC 80/10 life test program are described in Tables IV, V, and VI. If the data from Table IV is used along with the information from the reliability mathematics section, the +25°C MTBF for the SBC 80/10 Single Board Computer at a confidence level of 90% is 91,739 hours (failure rate = 1.09% per 1000 hours). A high temperature analysis was done concurrently with the +25°C program. As shown in Table VI, the
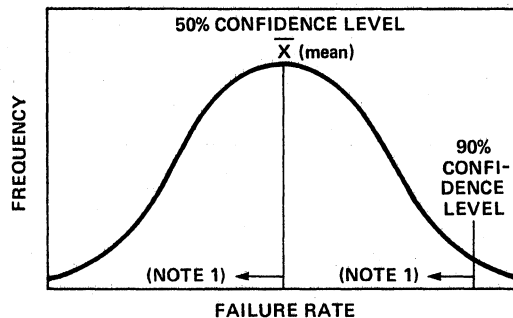
NOTE 1: ALL BOARDS AT A GIVEN CONFIDENCE LEVEL HAVE FAILURE RATE < THE FAILURE RATE AT THAT LEVEL

Figure 10. Failure Rate vs. Failure Frequency

SBC 80/10 Single Board Computer is shown to have an MTBF of 25,006 hours (failure rate = 3.99% per 1000 hours) with a confidence level of 90% when operated continuously at ambient temperatures of +55°C (specified maximum ambient temperature for SBC 80/10 computer operation).

Field Data

The SBC 80/10 Single Board Computer was introduced in early 1976. Based on the number of SBC 80/10 Single Board Computers shipped to date, a conservative estimation of actual unit running time provides an experience of more than 2,579,000 unit-hours of operation. Intel field service has experienced 18 SBC 80/10 computer failures during this operational period. This data yields a 90,845-hour, +25°C MTBF at a 90% confidence level (25°C operation assumed for all unit-hours). This correlates well with the results of the life testing program.

Table IV

LIFE TEST DATA EXTRAPOLATED TO 25°C

TEST TEMPERATURE	+25°C	+55°C	+70°C	TOTAL	FAILURES
Unit-Hours @ Temperature	24663	14296	20020	—	1
Equivalent Unit-Hours @ +25°C	24663	84624	256580	365867	1

Table V

LIFE TEST DATA EXTRAPOLATED TO 55°C

TEST TEMPERATURE	+55°C	+70°C	TOTAL	FAILURES
Unit-Hours @ Temperature	14296	20020	—	0
Equivalent Unit-Hours @ +55°C	14296	43345	57641	0

Table VI

LIFE TEST RESULTS

TEST TEMPERATURE	FAILURE RATE PER 1000 hours 90% UCL	MTBF, hours 90% UCL
+25°C	1.09%	91,739
+55°C	3.99%	25006

SUMMARY

As a result of the reliability tests and Intel's reliability program, the SBC 80/10 Single Board Computer has demonstrated an MTBF of 25,006 hours at 90% confidence level when operated at 55°C. Stated another way, if a system using an Intel® SBC 80/10 Single Board Computer were used 24 hours a day at an average ambient temperature of 55°C, there is a 90% probability that such a system will operate for almost 35 months before the system would fail due to an SBC 80/10 computer failure (over 10 years at +25°C operation). Of course, the system failure rate would be somewhat higher due to the failure rates of other elements in the system.

The SBC 80/10 Single Board Computer was introduced in January, 1976, and has become an industry standard. Intel's system of manufacturing microcomputer systems, coupled with extensive reliability monitoring and tests, has yielded a high reliability, low-cost product — the SBC 80/10 Single Board Computer.

APPENDIX

FLOWCHART OF RELIABILITY TEST PROGRAM

intel

RELIABILITY REPORT

RR-23

OCTOBER 1979

Intel® iSBC 86/12A™ Single Board Computer

Steve Carmichael
Reliability Engineer
Microcomputer Division

142646

Intel® iSBC 86/12A Single Board Computer Reliability Report

Contents

INTRODUCTION

This report presents reliabilty tests and supporting data for the Intel iSBC 86/12A Single Board Computer with the optional iSBC 300 32K Byte RAM Expansion Module and the iSBC 340 16K Byte EPROM/ROM Expansion Module. The iSBC 86/12A Single Board Computer is a complete computer system on a single printed circuit board; the CPU, system clock, read/write memory, read-only-memory, I/O ports, serial communications interface, priority interrupt logic, programmable timers, bus control logic and drivers all reside on the board. The optional memory expansion modules attach to the board, allowing the on-board RAM to be expanded by 32K bytes (for a board total of 64K bytes), and the on-board EPROM/ROM to be expanded by 16K bytes (for a board total of 32K bytes).

A complete reliability testing program has been implemented to ascertain the reliabilty level of the Intel iSBC 86/12A microcomputer and the iSBC 300 and iSBC 340 Expansion Modules. This report presents a review of this program, including a discussion of the test configuration utilized, the system exerciser/diagnostic software used, the number of hours of testing performed and the ambient test temperatures. High temperature testing was performed to accelerate product aging and to provide operational data on the reliability performance of the iSBC 86/12A microcomputer at elevated temperatures.

A full description of the quality assurance procedures that all of Intel's complete line of single board computers undergo prior to shipping is also included.

iSBC 86/12A™ BOARD DESCRIPTION

The iSBC 86/12A Single Board Computer is contained on a 6.75 by 12 inch (17.15 by 30.48 cm) printed circuit board. Figure 1 shows an overall block diagram of the board. Intel n-channel MOS LSI technology and bipolar TTL are the basis for a full computer on a single PC board. Such a single board approach is inherently more reliable than traditional multiboard implementations through minimization of inter-board connectors.

The heart of the iSBC 86/12A Single Board Computer is the Intel 8086 16-bit microprocessor (CPU). The 8086 CPU includes four 16-bit general purpose registers, which may also be addressed as eight 8-bit registers. In addition, the CPU contains two 16-bit pointer registers and two 16-bit index registers. Four 16-bit segment registers allow extended addressing to a full megabyte of memory.

The standard iSBC 86/12A microcomputer includes 32K bytes of read/write memory; connectors are provided to allow the addition of up to 16K bytes of programmable read only memory in 1K, 2K, or 4K byte increments. Intel EPROMs are used for programmable read only memory. Up to 32K bytes of additional on-board RAM and 16K bytes of additional EPROM can be added to the standard iSBC 86/12A board with the iSBC 300 and the iSBC 340 Expansion Modules, respectively.
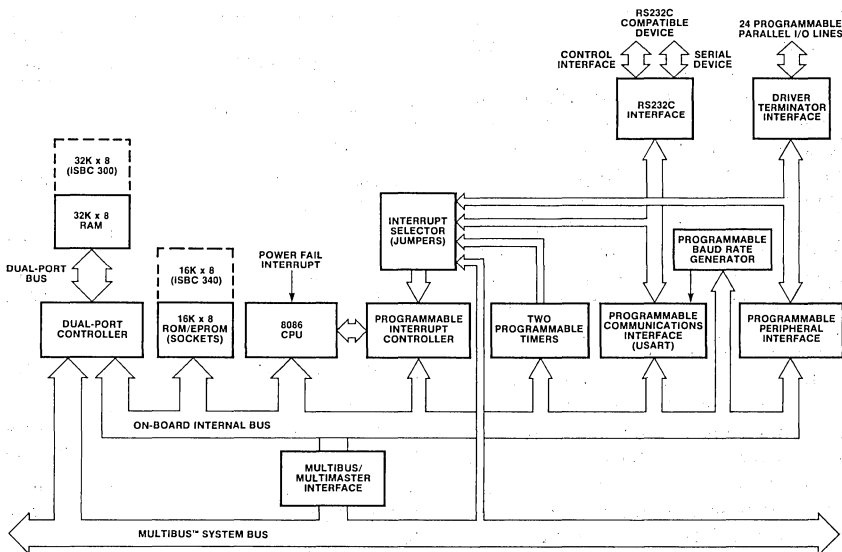
Figure 1. iSBC 86/12A™ Single Board Computer Block Diagram

One Intel® 8255A Programmable Peripheral Interface device supplies 24 programmable parallel I/O lines. These lines are routed to sockets on the board that are used to buffer lines programmed as outputs with TTL drivers or to terminate lines programmed as inputs with iSBC 901 or iSBC 902 resistor terminator packs.

For serial communication, an Intel® 8251A Universal Synchronous/Asynchronous Receiver/Transmitter (USART) is part of the iSBC 86/12A board. Additional circuitry provides jumper-selectable Teletypewriter (20 mA current loop) or RS232C compatible interfaces for the USART.

For further information on detailed operation of the iSBC 86/12A microcomputer, consult the *Intel iSBC 86/12A Single Board Computer Hardware Reference Manual.*

SINGLE BOARD COMPUTER QUALITY ASSURANCE

The reliability of a board level product like the Intel iSBC 86/12A Single Board Computer is a function of the quality of components used on the board, care in board design and fabrication, and the extent of testing performed on the product before shipment. An examination of each of these functions will provide an understanding of the Intel quality assurance program for micro-computer system products.

Component Quality Assurance

Standard Intel component quality assurance processing and 100% screening are applied to all Intel components used on the iSBC 86/12A microcomputer before they are assembled on the board. Once a component device has been quali-fied as reliable and transferred to systems level manufacturing, it is imperative that complete process controls are in effect to assure the continuation of high quality. Intel's component QA flow centers around a series of acceptance gates between process entities (see Figure 2) and detailed inspection within the processing areas at critical points. For example, in wafer processing, furnaces are routinely monitored for contamina-tion through the use of capacitance voltage measurements on test chips. Also, electrical tests such as breakdown strength measurements are performed on test patterns on each wafer. Routine high magnification scanning electron microscope examinations at critical process steps also provide important process control feedback. A final QA acceptance is performed on all lots prior to shipment to assembly locations.

Figure 2. Component QA Flow Diagram

Table 1 details the Intel component QA acceptance gates within the assembly flow and identifies the appropriate method employed per MIL-STD-883. During the test and finishing operation, QA maintains standards, test tape and calibration control of all production equipment. Final QA inspection is performed after the mark and pack operation.

Table 1. Component Assembly Flow

OPERATION	MIL-STD-883 METHOD
Piece Part Inspections	—
Scribe and Break	—
QA Acceptance	2010.1B
2nd Optical Inspection	2010.1B
Lead Bond	—
QA Acceptance	2010.1B
3rd Optical Inspection	2010.1B
Seal	—
Temperature Cycle	1010 (C)
QA Acceptance	1014 (B)
Fine Leak	1014 (A)
	(10^7 atm cc/sec)
QA Acceptance	1014 (C)
Gross Leak	1014 (C)
QA Acceptance	2009
Final Visual	2009

Board Assembly

Figure 3 shows the stringent cycle of inspection and testing each board undergoes before being shipped to the customer. After bare board manufacture, boards are inspected for raw board quality conformance (i.e., hole registration, drilling, laminate integrity, artwork registration, etc.), and tested for trace "shorts" and "opens".

Components for an assembly "kit" are then pulled together and made ready for the assembly operation. Visual inspection of each kit of components is then conducted. The kit is assembled onto the bare board and the assembled parts of the board are inspected prior to wave solder for proper location and after wave solder for soldering defects. Final assembly inspection insures the boards are ready for test.
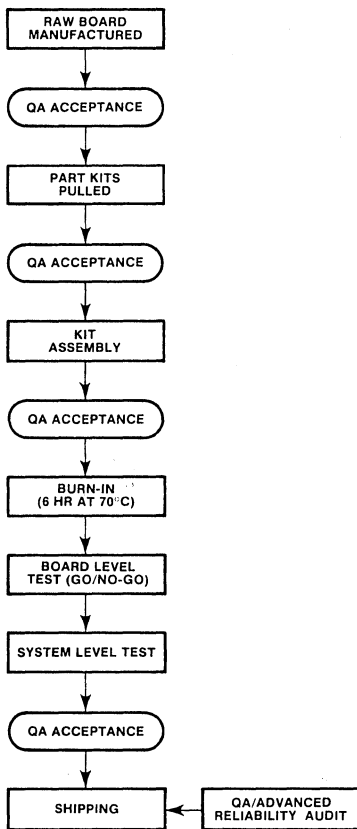
Board Production Testing

Manufacturing testing, though not a part of reliability testing per se, is very effective in eliminating "infant mortality" failure. These types of failures, by their nature, occur in the early life of a product. Each iSBC 86/12A microcomputer is pre-baked at 70°C prior to any testing. Figure 4 shows some of the ovens used in this process. The boards are allowed to cool and then exhaustively tested on a Teradyne L125 Digital Diagnostic Tester. Use of a special "bed of nails" vacuum test fixture insures that all critical board test points may be accessed by the test system. Figure 5 shows the test fixture for the iSBC 86/12A board, and Figure 6 shows one of the Teradyne test
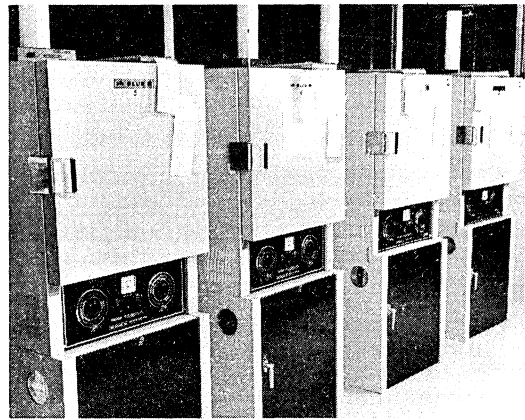
Figure 4. Ovens Utilized for Board Pre-Bake

Figure 3. Board Manufacture
and Test Flow/Screening
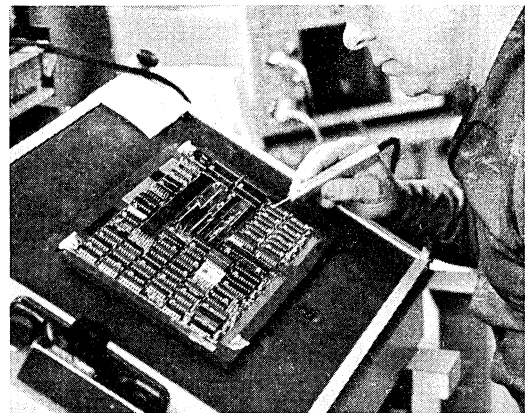
Figure 5. iSBC 86/12A™ Single Board
Computer Test Fixture

systems used in Intel microcomputer board-level testing. Failed components are replaced, and the boards are retested on the Teradyne test system.

When the boards have passed this thorough testing, they are then "system tested" to guarantee Intel® MULTIBUS™ interface integrity. This consists of installing a set of monitor and diagnostic program memory EPROMs into the four iSBC 86/12A EPROM sockets, applying all necessary voltages, and monitoring the iSBC 86/12A board's performance with a CRT terminal (see Figure 7). The *monitor/diagnostic exerciser* program tests the CPU, dual port memory, I/O interfaces, and MULTIBUS interface in a multi-processing environment. After test, the boards are inspected for possible test damage. Boards passing this final screening are then released by Intel Quality Assurance to Intel's standard product warehouse. Each product shipped is also subject to an audit. This audit consists of a complete review of documentation, test, functional performance, workmanship and handling.

Figure 6. A Teradyne Test System Utilized in iSBC 86/12A™ Board Testing

Environmental and Temperature Testing

Intel's Advance Reliability/Quality Assurance Department also performed a series of environmental and temperature tests on production versions of the iSBC 86/12A Single Board Computer, the iSBC 300 RAM Expansion Module and the iSBC 340 EPROM/ROM Expansion Module. These tests were designed to assure that the boards can withstand the worst conceivable physical and temperature conditions that might be found in a light commercial or industrial environment. A light commercial or industrial environment is defined as an area with moderate temperature and humidity, suitable for occupation by operating personnel, as for example an office or a manufacturing plant.

The temperature tests were performed in Intel's Environmental Testing Laboratory in Hillsboro, Oregon. VIKING LAB, an independent laboratory in Mountain View, California, was engaged to perform the environmental tests, which consisted of vibration, shock and humidity testing.

The iSBC 86/12A board was tested in two test configurations. In one configuration, two iSBC 86/12A boards were installed in an iSBC 604 cardcage and exercised in a master/slave multi-processing environment. The other configuration was identical except that an iSBC 300 Expansion Module and an iSBC 340 Memory Expansion Module were connected to each iSBC 86/12A board in the test configuration. When possible, each of the board configurations were tested concurrently. The programs to exercise the boards during the various environmental and temperature tests were contained on EPROMs installed on each iSBC 86/12A board.

Two parallel I/O ports were used as test monitors with specially designed LED fixtures. A CRT terminal connected to the serial I/O port was used to start the test and monitor the system testing.

Both boards in each board configuration were exercised over the MULTIBUS interface, which in turn exercised the Intel® 8289 Bus Arbiter. Dual port exchange was exercised as well between RAMs. Approximately 80% of the CPU's instruction set was exercised.

The EPROM on each board exercised and checked its resident RAM (and expansion module RAM when present) as well as the RAM (and expansion module RAM) on the companion board. This cross checking allowed maximum exercising of the boards. The exercise routines were then looped to provide continuous operation of the boards before, during, and following the various environmental and temperature tests.

Vibration

For the vibration tests, the two board configurations were tested on a horizontal vibration table in all three axes (X, Y and Z). On each axis the vibrations applied were cycled up and down between 10 and 55 Hz, with a total cycle time of 1½ hours. Each major resonance point was dwelled at for 15 minutes, during which time a strobe light was used to observe the relative movement between boards. If no resonance points were observed during the first half of the test cycle, the 15 minute dwell was performed at 55 Hz. At the completion of the test of each axis, a visual inspection of the boards was performed. The iSBC
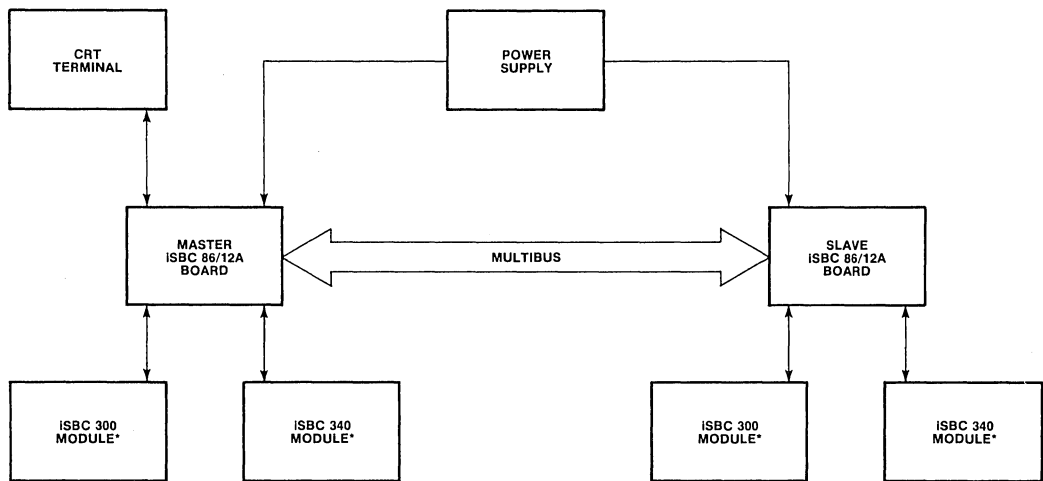
* Expansion modules removed in second test configuration.

Figure 7. Block Diagram of Test Set-Up PSI76

86/12A board, iSBC 300 and iSBC 340 modules passed the vibration test without failure. This test is intended to simulate an environment with low level vibrations such as those caused by a large industrial motor or machine, or as might be found in a mobile installation.

Shock

In testing the ability of the board configurations to withstand shock, the board configurations were subjected to 30g shocks for 11 μsec (½ sinewave). Shocks were applied to each of the six sides of the board configuration, 12 shocks per side. A performance test cycle was performed before, during, and following each shock; a visual inspection was performed at the completion of the test of each side. At the end of the shock test, a post-test performance test was performed. No visual damage was noted as a result of the shock tests to the iSBC 86/12A boards or the iSBC 300 or iSBC 340 modules, nor did the tests have any effect on the performance of the boards. This test simulates typical shipping conditions with commercial carriers and the kind of occassional shocks that occur from bumps or dropping a piece of equipment from a low height.

Humidity

Each board configuration was subjected to the following test cycle in a humidity chamber: (while operational)

1. Relative humidity and temperature of the chamber was ramped up to 80% and 25°C (taking approximately 1 hour).

2. Relative humidity was held at 80% for 2 hours.

3. Relative humidity and temperature are then ramped up to 95% and 40°C (taking approximately 1 hour).

4. Chamber is held at this state for 17 hours.

5. Relative humidity and temperature are ramped down.

This cycle was performed four times on each board configuration with no effect on the performance of the boards and no resulting board or component failure. These humidity conditions test the boards reliability not only at high humidity and temperature, but also accelerates the effects of moderate humidity over a long period of time.

Temperature

Three temperature related tests were performed on the board configurations: a non-operating test, an operating test, and a temperature vs. power supply voltage test. For the non-operating test, the board configuration was exercised, then placed in a –40°C environment for 2 hours. After removal from this environment, the board configuration was allowed to stabilize for 1 hour and then it was exercised again. This test was repeated at 75°C.

For the operating test, the board configuration was exercised at 55°C for 16 hours, turned off and allowed to stabilize for ½ hour at room temperature, then operated for 1½ hours at 0°C. These non-operating temperature tests are intended to simulate worst case storage or transportation temperatures.

For the temperature vs. power supply voltage test, the three power supply voltages (+5V, –5V and

-12V) are adjusted first to their +5% tolerance extreme and then to their -5% tolerance extreme. At each of these settings, the board configurations were exercised for 2 hour intervals both at 0°C and +55°C. These two procedures test the ability of the boards to continue operating in extremes of a light commercial or industrial environment and under the combined conditions of extreme temperature variation and poorly adjusted or unstable power supply voltages.

The iSBC 86/12A, iSBC 300 and iSBC 340 boards performed without problem through each of these temperature tests.

Summary

Having been subjected to these tests as a part of Intel's reliability testing program, the iSBC 86/12A Single Board Computer, the iSBC 300 32K Byte RAM Expansion Module, and the iSBC 340 16K Byte EPROM/ROM Expansion Module have demonstrated a high reliability/confidence level.

The reliability of these boards will be further monitored through continued reliability testing, Intel's production quality assurance program and the analysis of field reports. Evidence of failures will be distilled back into the engineering process and used in refining quality assurance testing procedures.

Reduce your μ C-based system design time by using single-board microcomputers

By George Adams
Electronic Design 3/February 1, 1978

Technology

Reduce your μC-based system design time by using single-board microcomputers. Assembled boards in the SBC-80 series offer stock answers to custom demands.

System designers eager to take advantage of the dramatically increased capabilities of micro-computers have been hindered two ways: Their production volumes have been too low to amortize software and hardware development costs effectively, or hardware subtleties and test requirements have confined them to fully assembled and tested computer subsystems. But now those obstacles are overcome with families of fully assembled and tested microcomputers and system-expansion boards like the Intel SBC-80 series. They are ready-to-use, flexible and inexpensive—prices range from just $195 to $825 in unit quantities.

The main members of the SBC-80 family are the 80/04, 80/05, 80/10A, 80/20 and 80/20-4 central-processor boards, with either an 8080A or 8085 microprocessor acting as the master CPU (Table 1). Most of the boards measure 6.75 × 12 in. and contain the CPU, clock, read/write memory, control ROM, I/O ports, serial communications interface and bus-control logic.

I/O interfacing is an area where design flexibility is essential to meet changing requirements efficiently. The programmable parallel and serial I/O structures of the boards make them versatile enough to do just that. What's more, upgrading system performance is easy thanks to the SBC-80 system bus, the Multibus, which permits modular performance expansion.

The Multibus provides a defined, standard interface between the SBC-80 single-board computers and expansion boards. As many as 16 SBC-80 family boards can simultaneously share the bus.

All in the SBC-80 family

As exemplified by the block diagram of the SBC-80/10A (Fig. 1), the SBC-80 microcomputer system has all that's needed for many applications. The SBC-80/10A is the oldest board in the family and has been widely imitated since it was one of the first "standardized" microcomputers commercially available.

The CPU section of the 80/10A board consists of

the 8080A CPU, the 8224 clock generator and the 8238 system controller. Capable of fetching and executing any of the 8080A's 78 instructions, the CPU section can respond to interrupt requests originating on and off the board. (For more about the 8080A, see "Microprocessor Basics, Part 2," ED No. 10, May 10, 1976, p. 84).

The system-bus interface section includes an assortment of circuits to gate the interrupt and hold requests, the ready signals, and a system-reset signal. Other circuits drive the various control lines. Two 8216s help drive the bidirectional data bus, and six 8226s drive the external system-data and address buses as part of the SBC-80/10A's Multibus interface.

The RAM section of the 80/10A consists of 1024 bytes of static MOS memory. For program storage, up to 8192 bytes of ROM can be mounted on the board in 1024-byte increments by means of a 2708 or 8708 EPROM, an 8308 mask-programmed ROM, or in 2048 byte increments via the 2716 EPROM or 2316 ROM.

A serial interface on the board uses an 8251 programmable universal synchronous/asynchronous receiver/transmitter to provide a serial-data channel. The serial port operates at programmable rates up to 38,400 baud (synchronously) or 19,200 baud (asynchronously) with a choice of character length, number of stop bits, and even, odd or no parity. On-board interfaces provide direct EIA RS-232 or teletypewriter current-loop compatibility.

Two 8255 programmable peripheral interface circuits provide 48 I/O lines for transferring data to or from peripheral devices. Eight already-committed lines have bidirectional drivers and termination networks permanently installed, so that they can be inputs, outputs or bidirectional (jumper-selectable). The other 40 lines are uncommitted. On-board sockets permit drivers and termination networks to be installed, as needed. Since software configures the I/O lines, I/O can be customized for every application.

The 80/10A also responds to a single-level interrupt that can originate from one of many sources, the USART, programmable I/O and two user-designated interrupt-request lines. When an interrupt is recognized, a Restart-7 instruction is generated, and the processor accesses location 38_H to get the starting address of the service routine.

System expansion and support are possible with a

George Adams, Product Line Manager, Single-Chip Micro-computers, Intel Corp., 3065 Bowers Ave., Santa Clara, CA 95051.

Note: Multibus, RMX-80, ICE and Intellec are registered trademarks of Intel Corp.

wide variety of alternate-source CPU, memory, and I/O boards (Tables 2 and 3). Up to 65,536 bytes of ROM, PROM or RAM can be accessed by one 80/10A. Expandable backplanes and card cages are also available to support multiboard systems.

Interfacing starts with the bus

Although the SBC-80/10A is a complete microcomputer system, it can be expanded readily or it can serve as a primary master controller for other microcomputer cards. The 80/10A has five edge connectors, three on the top of the board and two on the backplane, or bottom, side. Two of the "top" connectors, J_1 and J_2, serve as parallel I/O ports, while J_3 is a serial I/O port. All parallel I/O lines on the 50-contact J_1 and J_2 connector areas are paired with an independent signal/ground pin to permit alternate signal/ground wiring when using flat-cable interconnects. Serial port J_3 uses a 26-contact PC-edge connector to provide interfaces for both RS-232 and current-loop devices.

To communicate with other system-compatible boards, the 80/10A uses the 86-pin Multibus (P_1). To provide accessible test points, the 80/10A has a 60-pin edge connector (P_2). The control signals on the Multibus provide the real power and capability in control applications.

Of the 86 pins that make up the Multibus, 24 are assigned to power and ground, 16 to addressing, eight to bidirectional data, and 12 to signal and control
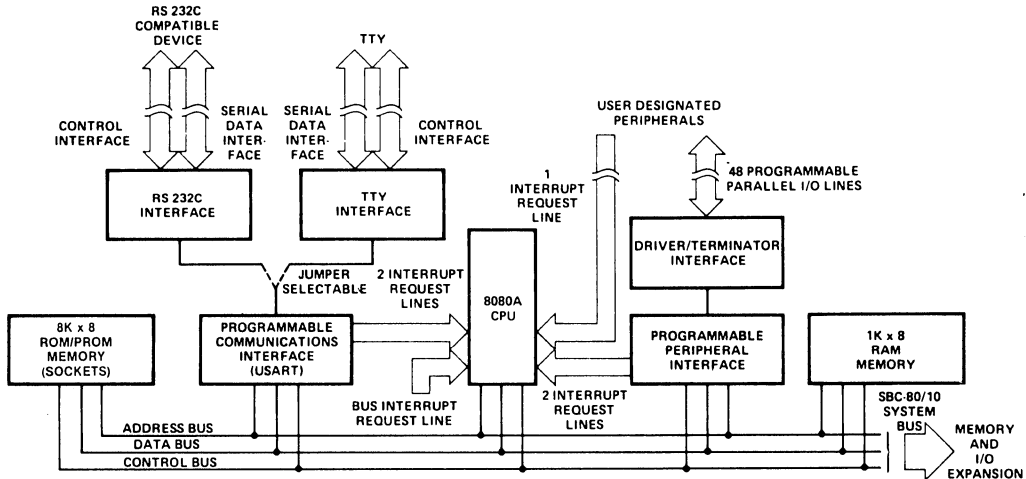
Interrupts originating from the Programmable Communications Interface and Programmable Peripheral Interface are jumper selectable.
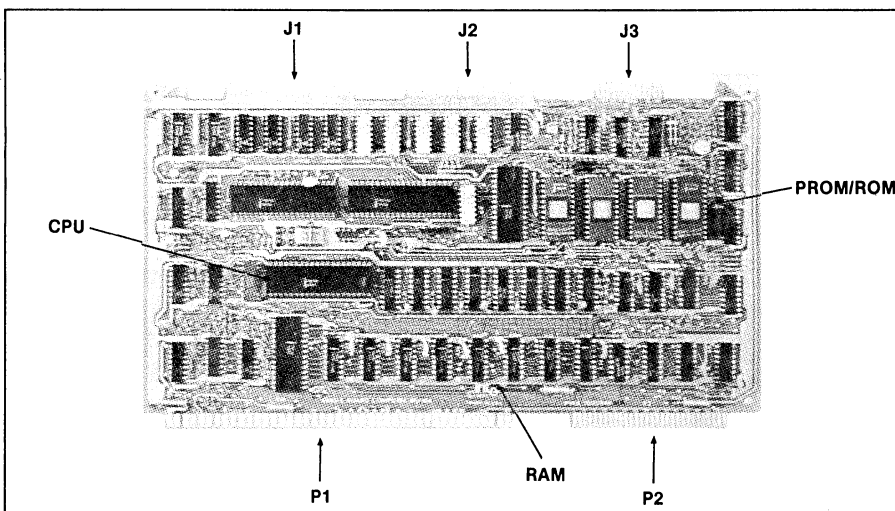
1. **Based on an 8080A μP, the 80/10A microcomputer has** a straightforward design suitable for general-purpose computing and control. The board has 48 programmable I/O lines and serial interfaces.
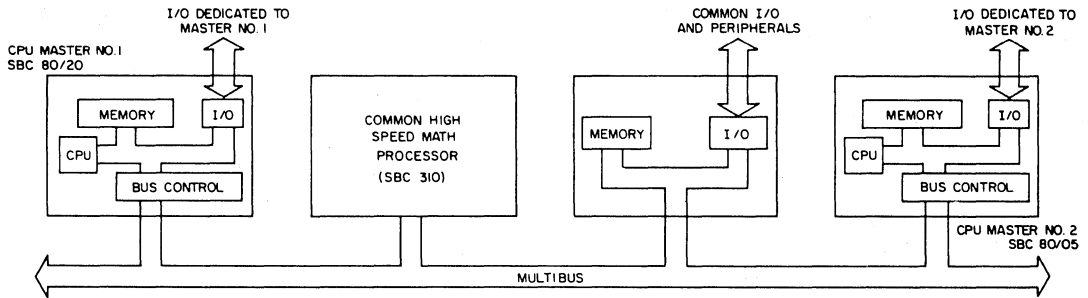
2. The Multibus interface for the SBC-80 CPU boards not only permits simultaneous multiprocessing, but also enables several processors to share the same bus and peripheral devices. Arbitration logic on the CPU boards decides which board gets on the bus first if several units simultaneously access the bus.

(these 12 are defined in Table 4). The remaining 26 pins are unassigned at this point. Higher capability SBC-80 products, though, are in development. These boards will use many of the unassigned lines (eight unassigned pins are allocated for additional bidirectional data lines). The remaining lines provide multilevel (eight) interrupt lines, various control lines and a multimaster, bus-arbitration control structure (Fig. 2). Address and data lines are three-state, and the interrupt and control lines are open-collector.

Boards using the Multibus have a master-slave relationship: A bus master—such as an SBC 80 CPU board, a DMA controller or a diskette controller—can control the command and address lines. Conversely, slave boards—such as a memory, I/O-expansion or mathematics boards—cannot control the bus.

Arbitration resolves priority disputes

In multimaster systems, the bus-arbitration logic uses the $\overline{CCLK}$ signal of the bus to provide a timing reference to help satisfy many simultaneous requests for bus control. As a result, different speed masters

can share resources on one bus. Actual transfers on the bus proceed asynchronously with respect to the bus clock. Once bus access is granted, single or multiple read/write transfers can proceed at up to 150 kbytes/s for CPU operations and up to 1 Mbyte/s for DMA operations. The bus has a bandwidth of 5 Mbytes/s so that future performance enhancements may be directly supported.

Both serial and parallel modes of bus-priority resolution are available. In the serial mode, up to three masters can share the system bus, with requests ordered on the basis of bus location. Each master on the bus notifies the next one down in priority when it needs to use the bus, and monitors the bus-request status of the closest higher-priority master. With an external priority network, up to 16 masters can share the bus.

The dual-bus nature of the Multibus permits each processor-based master within the system to retain its own local memory and I/O, which it uses for most operations. Such local operations occur entirely on the individual board and don't require the system bus.

In contrast to the dual bus architecture, all masters

Table 1. Comparison of SBC-80 CPUs

	SBC 80/04	SBC 80/05	SBC 80/10A	SBC 80/20	SBC 80/20-4
CPU	8085	8085	8080A	8080A	8080A
EPROM capacity (bytes)					
(with 2716)	4096	4096	8192	8192	8192
(with 2708)	2048	2048	4096	4096	4096
RAM (bytes)	256	512	1024	2048	4096
Programmable parallel I/O lines	22	22	48	48	48
Serial I/O capability	RS232C SID/SOD[1, 2]	RS232C SID/SOD[1, 2]	RS232C/TTY USART	RS232C[2] USART	RS232C[2] USART
Timers	1	1	0	2	2
Interrupt levels	4	4	1	8	8
Multibus interface	None	Multi-master	Single-master	Multi-master	Multi-master
Price (unit quantity)	$195	$350	$495	$735	$825

Notes: [1]Provided by 8085 CPU SID and SOD serial I/O lines. [2]Optional SBC 530 TTY interface is available.

Table 2. Additional SBC support boards

Function	Model	Description	Price (unit qty)
RAM	SBC 016	16 kbyte dynamic RAM	$ 825
	SBC 032	32 kbyte dynamic RAM	$1360
	SBC 048	48 kbyte dynamic RAM	$1860
	SBC 064	64 kbyte dynamic RAM	$2200
	SBC 094*	4 kbyte CMOS static RAM with 96 hour battery backup.	$ 795
EPROM	SBC 416	16 kbytes using 2708 type (1024 x 8) EPROM	$ 295
Digital I/O	SBC 508*	32 input lines/32 output lines, all buffered/terminated	$ 350
	SBC 517	48 programmable parallel lines with full buffering/termination options, full RS232C port, 1 ms real-time clock, and 8-line interrupt control	$ 400
	SBC 519*	72 programmable parallel lines with full buffering/termination options, real-time clock (interval is jumper selectable to 0.5, 1, 2, or 4 ms), and 8-level programmable interrupt control.	$ 395
Communications	SBC 534	Four programmable synchronous/asynchronous serial ports, each with: programmable baud rates, programmable data formats, programmable interrupt control, 16 RS232C buffered programmable parallel I/O lines configured as a Bell Model 801 automatic calling unit interface. Two programmable 16-bit interval timers (usable as real-time clocks), and software selectable loopback of serial ports for diagnostic use.	$ 650
	SBC 556*	48 optically isolated lines; 24 input 16 output, and 8 programmable (in/out), 8-level programmable interrupt control, and 1 ms real-time clock.	$ 395
Analog I/O	SBC 711*	16/8 (single-ended/differential) 12-bit a/d channels; user expandable on-board to 32/16 channels	$ 895
	SBC 724*	Four 12-bit d/a channels	$ 750
	SBC 732*	Combination analog I/O; same a/d capability as SBC 711 plus 2 d/a channels	$1125
Combination memory and I/O	SBC 104	8 kbytes capacity (sockets) using 2716 (2 k x 8) EPROM or 4 k using 2708, 4 kbytes dynamic RAM, 48 programmable parallel I/O lines, with full buffering/termination, as options. RS-232C port, a 1 ms real-time clock, and an eight-line interrupt control	$ 715
	SBC 108	Same as SBC 104, except has 8 kbytes of dynamic RAM	$ 815
	SBC 116	Same as SBC 104, except has 16 kbytes of dynamic RAM	$ 985
High-speed math	SBC 310*	High speed mathematics processor including floating-point capability. (32 bit).	$ 595
Peripheral control	SBC 201	Dual single-density diskette controller	$ 995
	SBC 202	Quad double-density diskette controller	$1290
DMA control	SBC 501	DMA controller, up to 1 MHz transfer rates	$ 450

*Requires +5 V only.

in multimaster/single-bus systems use the common bus for all instruction or data fetches or whenever data must be written to output devices or memory. Rapidly, then, the system bus becomes the bottleneck for over-all system throughput. Masters in SBC-80 systems only use the Multibus when data or instructions are resident in common, or global, memory or I/O. Since masters can request the Multibus simultaneously, on-board arbitration logic resolves any multiple contention.

Examine board performance

A look at the entire family of SBC-80 microcomputers reveals varied levels of performance. All five boards are inexpensive, but the most inexpensive is the 80/04, which costs $99 in 100-unit quantities, and is intended for stand-alone applications. To get the cost down, the board was designed to use the 8085 CPU and the 8155 RAM, timer and I/O circuit.

The 80/04 contains an 8085 CPU, 256 bytes of RAM, space for up to 4 kbytes of EPROM (two 2716 EPROMs, or two 2708 EPROMs), 22 programmable parallel I/O lines with sockets for buffer and termination options, a 14-bit programmable timer/event counter, and provision for an RS-232-C serial port using the 8085 SID/SOD serial interface. The board can also house an on-board +5-V regulator, so an unregulated voltage can be connected.

The next step up, the 80/05, has the same architecture and connector types and pinouts as the 80/04. Direct software compatibility is achieved with the same CPU along with the same RAM, ROM, I/O, and timer addressing. However, the 80/05 contains twice as much RAM as the 80/04. And since the 80/05 has the full Multibus multimaster interface, 80/05-based systems can be expanded with any of the Multibus-compatible boards from Intel or other suppliers.

The SBC-80/10A comes next. It provides more on-board memory and I/O for systems requiring expanded on-board resources. Based on the 8080A CPU, the board contains 1 kbyte of RAM, up to 8 kbytes of EPROM/ROM, 48 programmable parallel I/O lines, a full USART serial port with RS-232-C and tele-

typewriter interfaces, and a full Multibus interface (but only single-master capability; the board has no multimaster capability). Intended for single-CPU systems with only one other Multibus peripheral controller, the 80/10A can interface with such as the SBC-201 or SBC-202 single and double-density diskette controllers, or the SBC-501 DMA controller.

System designers requiring the same on-board I/O capability as the SBC-80/10A but with more RAM, more efficient real-time capability, and full multimaster Multibus control can go further up the ladder to the SBC-80/20 or SBC-80/20-4. These boards differ only in that the 80/20 contains 2 kbytes of RAM and the 80/20-4 contains 4 kbytes. Both boards can hold up to 8192 bytes of ROM or EPROM, handle up

to eight levels of prioritized interrupt, and share the Multibus in the multimaster mode. Either board has two programmable interval timers/event counters. Auxiliary power buses and memory-protect control logic on the board permit battery backup of the RAM.

Take advantage of interrupts and timers

Real-time applications frequently require that high-priority programs operate on the basis of external events, time-of-day, or elapsed time without impacting current background processing. These multiprogramming requirements are supported in the 80/20 and 80/20-4 by an eight-level programmable interrupt controller (PIC) and two programmable interval timer/event counters. The priority level of any event generating an interrupt request is assigned through jumper selection and the priority algorithm chosen by system software.

Any combination of interrupt levels may be masked by storing a single byte in the interrupt-mask register contained by the PIC, whose four software-selectable priority algorithms are described in Table 5. The PIC generates a unique memory address for each interrupt level. These addresses are equally spaced at intervals of 4 or 8 bytes (software-selectable). The resulting 32 or 64-byte block may begin at any 32 or 64-byte boundary in the 65,536-byte memory space. A single 8080A jump instruction at each of these addresses then provides linkage to locate each interrupt service routine independently anywhere in memory.

The two programmable timers may be used to generate real-time clocks by requesting periodic interrupts through the PIC, so that the CPU is free to handle numerous other system-timing and control functions. The outputs and gate/trigger inputs of the timer/counters can be routed via jumpers to the PIC, the I/O driver/terminators, or the programmable parallel I/O.

Seven software-selectable timing/counting functions are available. Either timer may be set to act as a rate generator (divide-by-N counter), a square-wave
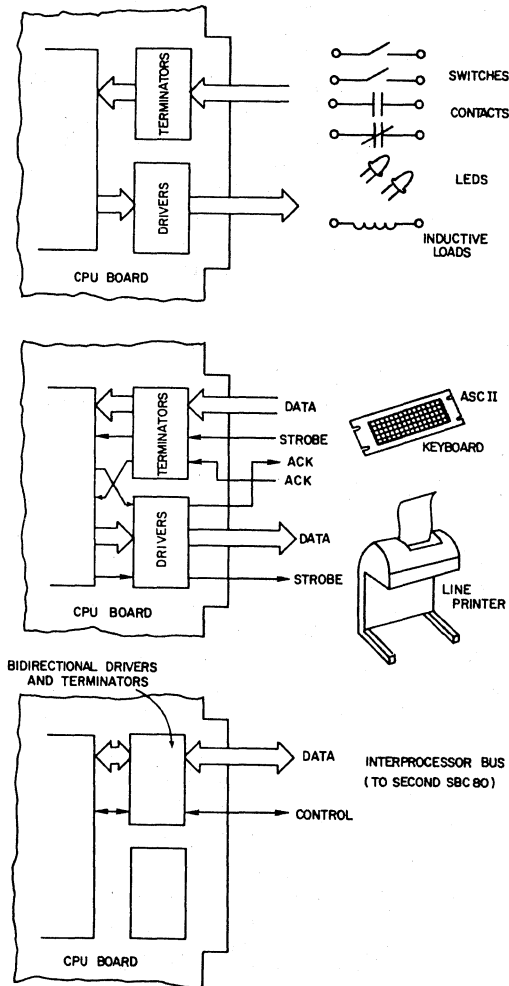
3. **Programmable I/O lines from the SBC-80 parallel interfaces** can be set so that they are individually programmable as inputs or outputs (a), byte-programmable as inputs or outputs with handshaking (b), or bidirectional on a byte-programmable basis (c).
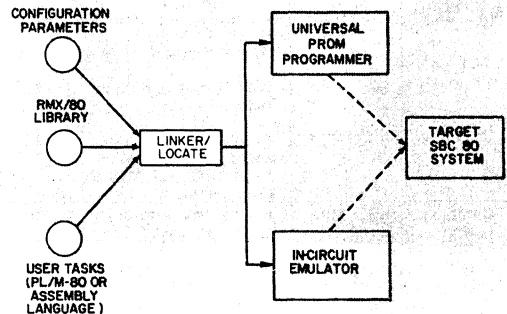
4. **By using the RMX-80 executive** and the library of often-used subroutines, program development can be simplified since the subroutines are modular and can be linked together, then checked out in a system prototype.

Table 3. Non-Intel SBC-compatible boards

Manufacturer	Multibus CPU boards	RAM boards	Core memory boards	PROM/ROM boards	Analog boards	AC power control boards	IEEE-488 bus	Tape interface	Floppy-disc controller	Communications interface	Video boards	Math processors	Breadboards	Circle no.
ADAC Corp., 118 Cummings Park, Woburn, MA 01801.(617) 935-6668					•									451
Ampex, Memory Products Div., 200 N. Nash St., El Segundo, CA 90245.(213) 640-0150			•											452
Analog Devices, Route 1 Industrial Park, P.O. Box 280, Norwood, MA 02062.(617) 329-4700					•									453
Augat Inc., 33 Perry Ave., P.O. Box 779, Attleboro, MA 02703.(617) 222-2202													•	454
Burr-Brown, International Airport Industrial Park, P.O. Box 11400, Tucson, AZ 85734.(602) 294-1431					•									455
Cybernetic Microsystems, 2460 Embarcadero Way, Palo Alto, CA 94303.(415) 321-0410					•					•				456
Data Translation Inc., 23 Strathmore Road, Natick, MA 01760.(617) 655-5300					•									457
Datacube Corp., 25 Industrial Park, Chelmsford, MA 01824. (617) 256-2555											•			458
Datel Systems Inc., 1020 Turnpike St., Building S., Canton, MA 02021.(617) 828-8000					•									459
Digidata Corp., 8580 Dorsey Run Road, Jessup, MD 20794.(301) 498-0200							•							460
EDAC Corp., 1417 San Antonio Ave., Alameda, CA 94501.(415) 521-6600											•			461
Electronic Engineering & Prod. Services, TE. #2, Louisville, TN 37777. (615) 984-9640						•								462
Electronic Solutions, 7969 Engineer Rd., San Diego, CA 92111.(714) 292-0242	•	•		•										463
Garry Mfg. Co., 1010 Jersey Ave., New Brunswick, NJ 08902.(201) 545-2424													•	464
Hal Communications Corp., Box 365B, 807 E. Green St., Urbana, IL 61801.(217) 367-7373										•				465
Iasis, 815 W. Maude Ave., Sunnyvale, CA 94086. (408) 732-5700	•													466
ICOM, 6741 Variel Ave., Canoga Park, CA 91303.(213) 348-1391									•					467
Megalogic Corp., 9650 National Road, Brookville, OH 45309.(513) 833-5222								•						468
Micro Memories Inc., 9438 Irondale Ave., Chatsworth, CA 91311.(213) 998-0700			•											469
Microtec, P.O. Box 60337, Sunnyvale, CA 94088.(408) 733-2919				•										470
Monolithic Systems Inc., 14 Inverness Drive, East, Englewood, CA 80110.(303) 770-7400	•	•												471
National Semiconductor, 2900 Semiconductor Drive, Santa Clara, CA 95051.(408) 737-5000	•													472
North Star Computers Inc., 2465 Fourth St., Berkeley, CA 94710.(415) 549-0858												•		473
The Thomas Engineering Co., 1201 Park Ave., Emeryville, CA 94608.(415) 547-5860										•				474
Vector Electronic Products, 12460 Gladstone Ave., Sylmar, CA 91342.(213) 365-9661													•	475
Zia Tech., 10762 La Roda Drive, Cupertino, CA 95015.(408) 996-7082							•							476

generator, a programmable retriggerable one-shot, or software or hardware-triggered strobe. One of the timers can be jumper-selected as an event counter, and either can generate an interrupt after a specified interval or after a specified number of events.

The programmability of each on-board timer allows timing intervals from approximately 2 μs to over 60 ms. But the two timers may be cascaded to provide intervals greater than 1.1 hour, in 1.86 μs increments. In the event counter mode, external event rates up to 1.1 MHz may be counted.

Flexible I/O, a must for any system

All SBC-80 microcomputers provide 22 or 48 programmable parallel I/O lines that, grouped as 8-bit ports, are fully programmable to allow enough flexibility to handle any changes in system interfacing. Programmability is permitted through data direction, control mode, interrupt handling, and buffer/termination. The I/O configuration for a specific application is selected through software initialization of the parallel I/O control logic, jumper selection of control/interrupt line routing, and the particular buffer and termination devices chosen.

Fig. 3 illustrates the basic modes of operation that may be selected by software to meet application requirements. Mode 0 is used for slow-to-medium-speed interfacing where immediate handshake response or interrupt generation is not needed. This mode is extremely useful for interfacing to inputs such as switches or outputs such as LED indicators or numeric displays.

Mode 1 provides handshaking lines required for many medium to high-speed peripherals. A typical output function could be a line printer; an input device could be an encoded keyboard or paper tape reader.

In addition, the 80/10A and 80/20 have Mode 2, a bidirectional data/control structure. This interface may provide, for example, a communication link between parallel processors.

The SBC-80 I/O structure also permits multiple options for output buffering and input termination. TTL drivers with 16 to 48 mA of drive can be used, and input lines may be terminated to minimize the impact of noise and cable disconnects. Any of the TTL drivers (four outputs) or input terminators (for inputs) listed in Table 6 may be inserted into sockets to provide proper buffering or termination.

Like the design flexibility of the SBC-80 parallel I/O structure, the serial I/O structure allows interface characteristics to be revised rapidly through software, jumper, and buffer changes. Besides the SBC-80/10A, the 80/20 and 80/20-4 contain the USART serial channel. These boards provide RS-232 interfaces, but the SBC-80/10A also has a teletypewriter current-loop interface. Synchronous/asynchronous mode, data format, control-character format, and parity are all under program control. So is baud rate on the 80/20 and 80/20-4. Baud rate is jumper-selectable on the

Table 4. Multibus control signals

AACK	Advance-acknowledge signal, used in 8080A-based systems. It is sent to the SBC-80 board by a memory bank in response to a memory-read command, allowing the memory to complete the access without requiring the CPU to wait.
BCLK	Bus clock, used to synchronize bus-control circuits on all master boards. It has a period of 101.725 ns (9.8304 MHz) and a 30 to 70% duty cycle. The signal may be slowed, stopped or single-stepped.
BPRN	Bus-priority-input signal, used to indicate to the master that a higher-priority master board wants to use the system bus. When brought high, the signal suspends processing activity and places line drivers of the master in a standby mode.
BUSY	Bus-busy signal, a bidirectional control line that allows control and monitoring of the Multibus in multimaster systems. As an output from a bus master, BUSY indicates the bus is being used by the board. It prevents all other master boards from gaining control of the bus. Each master monitors BUSY as an input to determine current Multibus usage status.
CCLK	Constant clock, used to provide a 9.8304-MHz clock signal for optional memory and I/O expansion boards. CCLK coincides with BCLK and has a period of 101.725 ns and a 30 to 70% duty cycle.
INIT	Initialize signal, used to reset the entire system to a known internal state.
INTR1	Interrupt input, used to interrupt the processor via an externally generated interrupt request.
IORC	I/O-read command, a signal generated by the master to indicate that the address of an input port has been placed on the system-address bus and that the data at that input port are to be read and placed on the system-data bus.
IOWC	I/O-write command, a signal generated by the master to indicate that the address of an output port has been placed on the system-address bus and that the contents of the system-data bus are to be output to the addressed port.
MRDC	Memory-read command, a signal generated by the master that indicates that the address of a memory location has been placed on the system-address bus. It specifies that the contents of the addressed location are to be read and placed on the system-data bus.
MWTC	Memory-write command, a signal generated by the master to indicate that the address of a memory location has been placed on the system-address bus. It causes information on the data bus to be written into the addressed memory location.
XACK	Transfer-acknowledge signal, an input signal to the master board from an external memory location or I/O port to indicate that a specified read or write operation has been completed.

80/10A CPU board.

The synchronous and asynchronous nature of the serial interface makes it compatible with virtually every standard serial data-transmission technique used today (including IBM's Bi-Sync). This allows multiple SBC-80 boards to be interconnected as a distributed-processing network. The resulting task segregation or redundancy (or both) significantly improves both system performance and reliability.

Two jumper-selectable interrupt requests may be generated automatically by the serial interface. One occurs when a newly received character is ready to be loaded into the CPU (receive-channel buffer is full). The other occurs when new data are ready to be transmitted to the remote device (transmit-data buffer is empty).

Both the SBC-80/04 and 80/05 provide serial I/O capability through the serial input data (SID) and serial output data (SOD) functions of the 8085 CPU. These functions are controlled by software executing the 8085 read-interrupt mask (RIM) and set-interrupt mask (SIM) instructions.

For systems requiring many serial channels, the SBC-534 communications-expansion board provides four USART channels with RS-232-C and optically isolated current-loop interfaces, programmable interrupt, timing, baud-rate control, and a Bell 801 Auto-Call unit interface.

Expand the system via the Multibus

The SBC-80 family is gaining not only in popularity but in support for its Multibus as more and more companies offer SBC-compatible boards. Intel now provides high-speed mathematics, RAM, EPROM, mass storage, digital I/O, combination memory and I/O, serial communications, and analog-I/O expansion boards.

For applications requiring fast, high-precision number crunching, the SBC-310 math unit acts as an intelligent slave to perform floating-point and fixed-point mathematics. A processor uses the 310 by passing parameters to it along with a command byte to select the desired operation from the SBC-310's 14 instructions. The repertoire includes 32-bit floating-point (single-precision) addition, subtraction, multiplication, division, squaring, square root, comparisons, and tests; 16-bit fixed-point multiply, subtract, extended divide, and extended compare; and conversion from fixed to floating point or vice versa.

A completed operation may be signaled either by the math unit via an interrupt or by the host processor's polling the "operation complete" flag in the unit's status register. The result may be retrieved at this point or left in the 310's accumulator for further use.

In addition, the 310 provides control circuitry so that it may be treated as a "shared resource" among several CPU boards.

Two diskette options are available for mass storage.

Table 5. Programmable interrupt modes, SBC-80/20-4

Mode	Operation
Fully nested	Interrupt request line priorities fixed at 0 as highest, 7 as lowest.
Autorotating	Equal priority. Each level, after receiving service, becomes the lowest priority level until next interrupt occurs.
Specific priority	System software assigns lowest priority level. Priority of all other levels based in sequence on this assignment.
Polled	System software examines priority-encoded system interrupt status via interrupt status register.

Table 6. Line drivers and terminators

Line drivers		
Driver	Characteristic	Sink current (mA)
7438	I,OC	48
7437	I	48
7432	NI	16
7426	I,OC	16
7409	NI,OC	16
7408	NI	16
7403	I,OC	16
7400	I	16

Note: I = inverting; NI = noninverting; OC = open collector
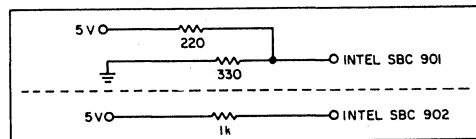
The SBC-201 diskette controller provides full control for one or two single-density diskette drives and acts as a programmable slave to masters on the Multibus. All diskette information is stored in the IBM soft-sectored format. For systems requiring greater storage capacity, the SBC-202 provides full control for up to four double-density diskette drives. Thus, 2 Mbytes of mass storage may be added to SBC-80-based systems for each SBC-202 plugged into the bus.

Digital I/O may be expanded using any of three Intel boards. The SBC-519 provides 72 programmable parallel I/O lines as well as interrupt handling and a real-time clock.

The 519's clock can interrupt the appropriate CPU periodically so that the CPU can monitor system-I/O status. High-speed I/O events can gain the CPU's attention via interrupts. The SBC-517 combination I/O board and the SBC-104, 108 and 116 combination memory and I/O boards offer 48 programmable parallel lines, a full RS-232 USART serial channel, interrupt handling and a 16-ms real-time clock. The

Table 7. RMX-80 routine library

RMX/80 module	Function
Nucleus (executive)	Provides basic capabilities (concurrence, priority, and synchronization/communication) found in all real-time systems.
Terminal handler	Provides real-time asynchronous I/O between an operator's terminal and tasks running under the RMX/80 executive, includes a line-edit feature similar to that of ISIS-II (supervisory system on the Intellec development system) and type-ahead facility.
Diskette file systems	Diskette driver and file management capabilities, allows user to load tasks into the system and to create, access, and delete files in a real-time environment without disrupting normal processing. File formats compatible with ISIS-II for both single and double-density systems.
Free space manager	Maintains a pool of free RAM and allocates memory out of the pool upon request from a task; reclaims memory areas when no longer needed.
Debugger	Specifically designed for debugging software running under the RMX/80 executive; used by linking it to an application program or task. Thus, it can be run directly from the single-board computer's memory.
Math handler	Provides full control and communication for SBC 310 math board for high-speed fixed and floating-point math functions.
Analog interface handler	Provides real-time control for SBC 711, 724, and 732 analog I/O expansion boards.

104, 108 and 116 also hold up to 8 kbytes of EPROM, along with 4, 8 or 16 kbytes of RAM, respectively.

For systems geared to especially noisy environments, the SBC-556 provides 48 optically isolated I/O lines, which are configured as 24 input lines, 16 output lines, and eight programmable-I/O lines. The user fixes the optical-isolation characteristics according to his exact system requirements by installing the opto-isolators and current-limiting resistors of his choice into the board sockets. Input voltages up to 48 V, output lines up to 30 V and currents up to 60 mA may be interfaced.

Of course, many more RAM, ROM, communications and interface options are available. But for systems to come together quickly during development, there must be some standardized operating software to provide some of the most fundamental system routines.

System software: the glue that binds

Where the Multibus provides a standard hardware structure, RMX-80, a real-time multitasking executive supplies a modular software framework. With RMX-80, routines don't have to be developed for task synchronization, priority resolution and peripheral control (printers, terminals, diskettes, etc.). Versions are available for the SBC-80/20, 80/20-4 and 80/10A CPU boards.

Critical projects can be completed rapidly because RMX-80 provides major portions of the software requirements for many real-time systems. For example, the diskette file-extension software of the RMX-80 program may be linked into the system software. Thus, users can immediately have a diskette file structure with facilities to open and close files, create and delete files, read or write files sequentially or randomly (read function may be used for initial program load, if desired), or allocate file storage dynamically on single or double-density diskettes.

The compactness of RMX-80—the entire executive resides in 2 kbytes of ROM—reduces memory requirements and eliminates the need for bootstrap-program loading. All RMX-80 operations are based on individual tasks. A task is a program with unique data and stack that operates asynchronously with other such programs in the system.

Basically, the RMX-80 is a library of "standard" routines (Table 7), such as an analog-interface handler and a terminal handler. Fig. 4 illustrates how to develop software by selecting appropriate RMX-80 modules, then locating and linking them with particular software tasks on an Intellec microcomputer development system. In addition, a debugger module
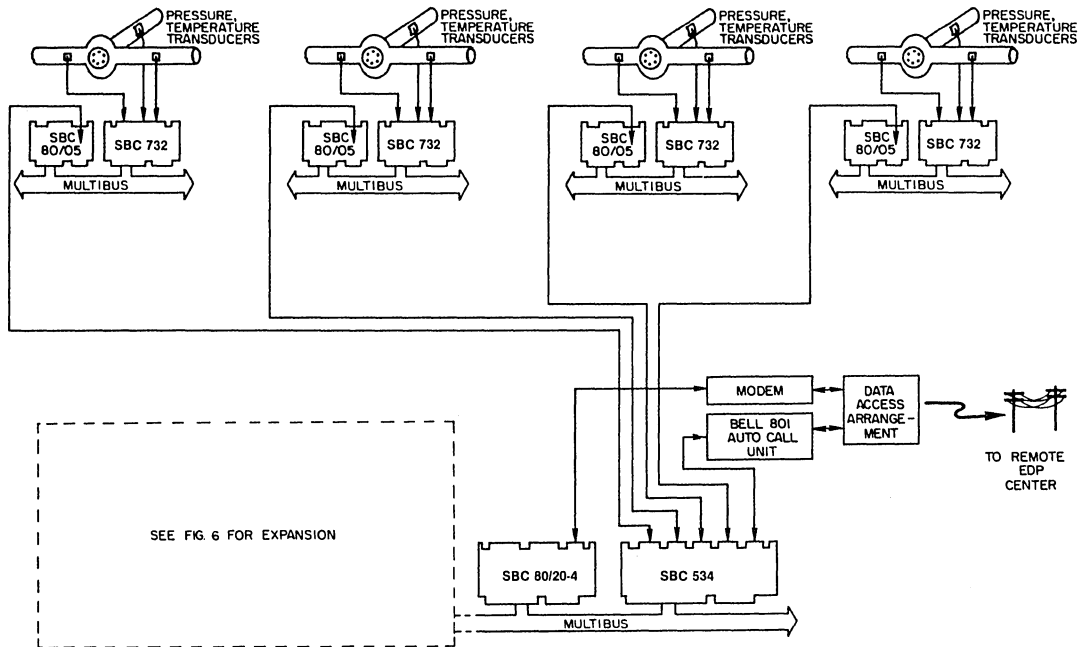
5. This possible SBC-80 system configuration uses four SBC-80/05s to monitor and control pipeline parameters and feed data back to a master controller, an SBC-80/20-4. The master controller sends data back to a host system.

in the RMX-80 speeds real-time system development.

The executive accesses system resources according to task priority, intertask communication, interrupt-driven control for standard devices, real-time clock control, interrupt handling, and other optional functions. In all, there are 255 separate task-priority levels, and since multiple tasks may share the same level, the actual number of tasks is limited only by memory size.

Develop programs with the Intellec

The Intellec and its ICE-80 and ICE-85 in-circuit emulators help minimize the time required to develop software and hardware. Standard Intellec stand-alone software includes resident macroassemblers for the 8080A and 8085 CPUs, a text editor, and a system monitor/debugger. As a result, programs can be assembled, loaded, edited, executed, and debugged.

ICE diagnostics can significantly reduce program development and debug time. Break points may be set on user-specified memory-read or write operations, I/O read or write operations, or user-defined extension parameters. Programs can be single-stepped to check operating conditions and performance.

PL/M-80 is the high-level systems-programming language. The optional Intellec-resident PL/M compiler provides the ability to program in this natural, algorithmic language, so there is no need to manage register usage or to allocate memory. PL/M programs
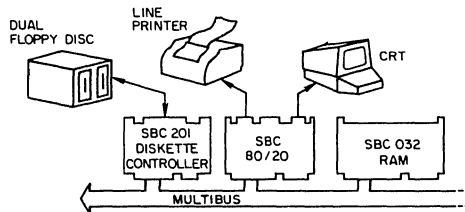
6. Expanding the pipeline monitor/controller system is as simple as plugging more cards into the Multibus and altering the software. By adding another SBC-80/20 to the master controller, some local processing can be done and a local CRT and high-speed printer can be added as well as RAM and diskette-memory space.

may be linked to assembly-language programs to hasten product development further.

A relocatable macroassembler residing on the Intellec translates symbolic assembly language into 8080 or 8085 machine code and permits the use of relocatable and linkable object code. With full macro capability, similar sections of code needn't be written over and over.

Intellec options include a diskette operating system, ISIS-II, with diskette controller, single or dual diskette

drives and ISIS-II software. ISIS-II provides all the facilities for producing and handling relocatable code, including a relocating macroassembler, relocating loader and a linker to help link separately compiled or assembled programs.

Apply the SBC boards to real use

To get an idea of the SBC 80 family's capabilities, examine the application shown in Fig. 5. In this case, a remote control/monitoring section of a pipeline supervisory control system grows with increasing requirements for additional local throughput and processing capability.

Four SBC-80/05s act as remote pipeline monitors/controllers. Each unit monitors various contact closures (limit switches, relays, etc.) and a hex keypad, with a subset of its own I/O lines programmed as inputs. Contact debounce is performed in software. Other digital I/O lines on each SBC-80/05 act as output lines to drive a numeric display and various control relay coils.

Analog-control lines are interfaced with an SBC-732 combination analog-I/O board. Transducers indicating temperature and pressure drive analog inputs, and analog outputs drive valves. Flow rate is determined in software by manipulating differential pressure data available from pressure transducers.

The four 80/05s are linked serially to a remote SBC-80/20-4-based data concentrator. An SBC-534 communications expansion board provides four RS-232-C serial channels, each interfacing directly with one of the four 80/05-based pipeline monitor/controllers. The 80/20-4 periodically queries each monitor to determine its current status. The concentrator also relays control commands from a host computer controlling the entire pipeline. The 80/20-4's own RS-232-C serial channel provides the interface for this high-speed synchronous link to the host CPU.

The 80/20-4 can contact the host CPU with the Bell 801 automatic calling-unit interface on the SBC-534. The synchronization and control of communication between the four 80/05s and the host are handled by RMX-80 on the 80/20-4.

The 80/20-4 system can be expanded to provide local processing capability, as shown in Fig. 6. Here, another 80/20 is added as a second master on the Multibus to provide control for a local CRT and high-speed printer, and to provide local processing capability.

An additional 32 kbytes of RAM are furnished by an SBC-032 RAM-expansion board. A third master, an SBC-202 dual-density diskette controller, can also be added to the Multibus, along with two double-density diskette drives. Communication between the two 80/20s is handled via user-written intermaster message tasks.■■

Design Motivations for Multiple Processor Microcomputer Systems

George Adams and Thomas Rolander
Microcomputer Applications

DESIGN MOTIVATIONS FOR MULTIPLE PROCESSOR MICROCOMPUTER SYSTEMS

Design decision factors involved in developing multiple processor microcomputer systems include means of minimizing contention for system bus utilization. System applications detail the appropriate hardware and software considerations as related to single-board computers in a multimaster bus structure

George Adams and Thomas Rolander   Intel Corporation, Santa Clara, California

Large-scale integrated circuit technology has reduced the cost of central processors to such a low level that the previously avoided concept of applying multiple processors to meet system performance requirements has now become an attractive and viable alternative. Several key benefits accrue from such an approach. In addition to enhanced system performance (throughput), improved system reliability, and improved system realtime response, modular system expansion capabilities may be realized. Although designing such systems "from scratch" with microprocessor component families can be a complex system design task with many subtle pitfalls which can inhibit efficient system operation, the advent of second generation single-board computers, such as the Intel® SBC 80/05 and 80/20, has allowed multiple processor microcomputer systems to become off-the-shelf products.

Motivation and Design Concepts

Discussion of the benefits of multiple processor structures in system applications will provide an understanding of the motivation for this implementation approach in system design. A primary objective addressed through multiple processor approaches is enhanced system performance and throughput. Enhanced performance is achieved through partitioning of overall system functions into tasks that each of several processors can handle individually.

In general, as the number of individual tasks any given processor must handle is reduced, that processor's response time to new requests for service will be reduced. A well planned multiple processor bus structure will allow new processors to be added to the system in modular fashion. When new system functions (ie, more peripherals) are added, more processing power can be applied to handle them without impacting existing processor (master) task partitioning.

As used here, a "master" is any element existing on the system bus that may take control of the bus (ie, assert address and control lines). Typical examples include processors and direct memory access (DMA) controllers that address memory and input/output (I/O) locations resident on the bus. "Slave" elements include passive functions on the bus, such as memory or non-DMA I/O interfaces. Note that although slaves may possess intelli-
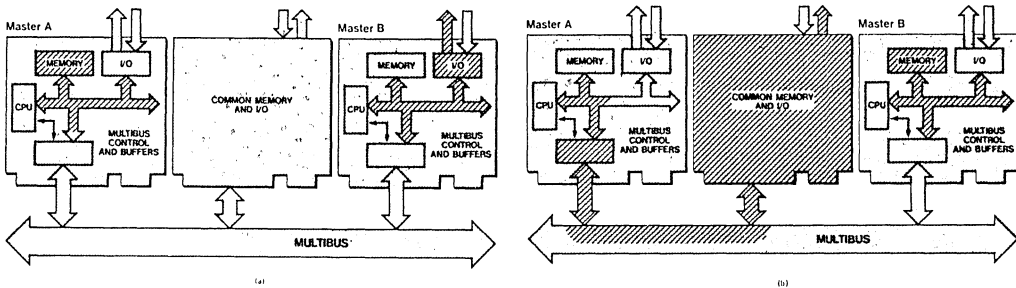
Fig 1 Multiple processor bus structure. Dual onboard/offboard structure of MULTIBUS allows each master to use its own memory and I/O without utilizing common system bus (a). Only when a master requires access to common memory or I/O does it use the bus (b). Note that other masters may continue onboard operations simultaneously
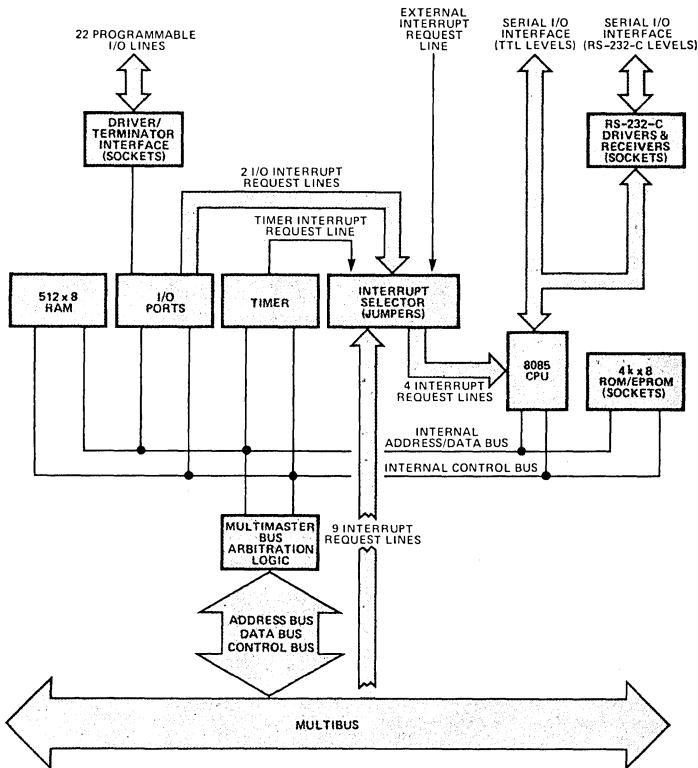
Fig 2 SBC 80/05 block diagram. SBC 80/05 is a full microcomputer on a single PC board. It provides 8085 CPU plus RAM for program or data storage, EPROM/ROM for program storage, interval timer, programmable parallel I/O (22 lines), serial I/O, and full MULTIBUS multimaster control logic

gence (eg, an onboard processor), they are not bus "masters" unless they can control the system bus.

Hardware Considerations

Hardware considerations must be thoroughly evaluated in any multiple processor bus structure. These factors are described in detail around a specific implementation of such a structure, the Intel[R] MULTIBUS[TM], which supports multiple processor systems with its multi-master bus structure.

Bus Architecture

One architectural option open to the system designer is that of a multiple master/single bus structure. Under this partitioning, every master utilizes the common bus data path to fetch instructions or data from memory, read data from input devices, or write data to output devices or memory. Therefore, the common system bus rapidly becomes the bottleneck for overall system throughput, and fast DMA transfers can easily approach the full bandwidth of the bus during block transfers so that all other masters must idle for extended periods.

Such performance constraints can severely limit total system performance.

System bus utilization may be minimized through implementation of an alternate dual-bus structure as shown in Fig 1. Each processor-based master within the system retains its own local memory and I/O that it utilizes for most operations. Such local operations occur totally on the individual board and do not require the system bus. This greatly reduces the service request frequency by each master requiring use of the system bus. Such a dual-bus structure is implemented on the SBC 80/05 and 80/20 single-board computers, as shown in Figs 2 and 3, respectively, with the multi-master system bus (MULTIBUS).[1,2]

Access to the system bus is requested only when a global (resident on the bus and accessible by multiple masters) memory location or I/O device is referenced during an instruction execution cycle. Local/global (on-board/offboard) distinction is defined through the value of the physical address referenced. If it lies within the address range of onboard memory or I/O, no bus request is made. Only when the address references a global

_____

Intel® and MULTIBUS™ are trademarks of Intel Corp, Santa Clara, Calif.
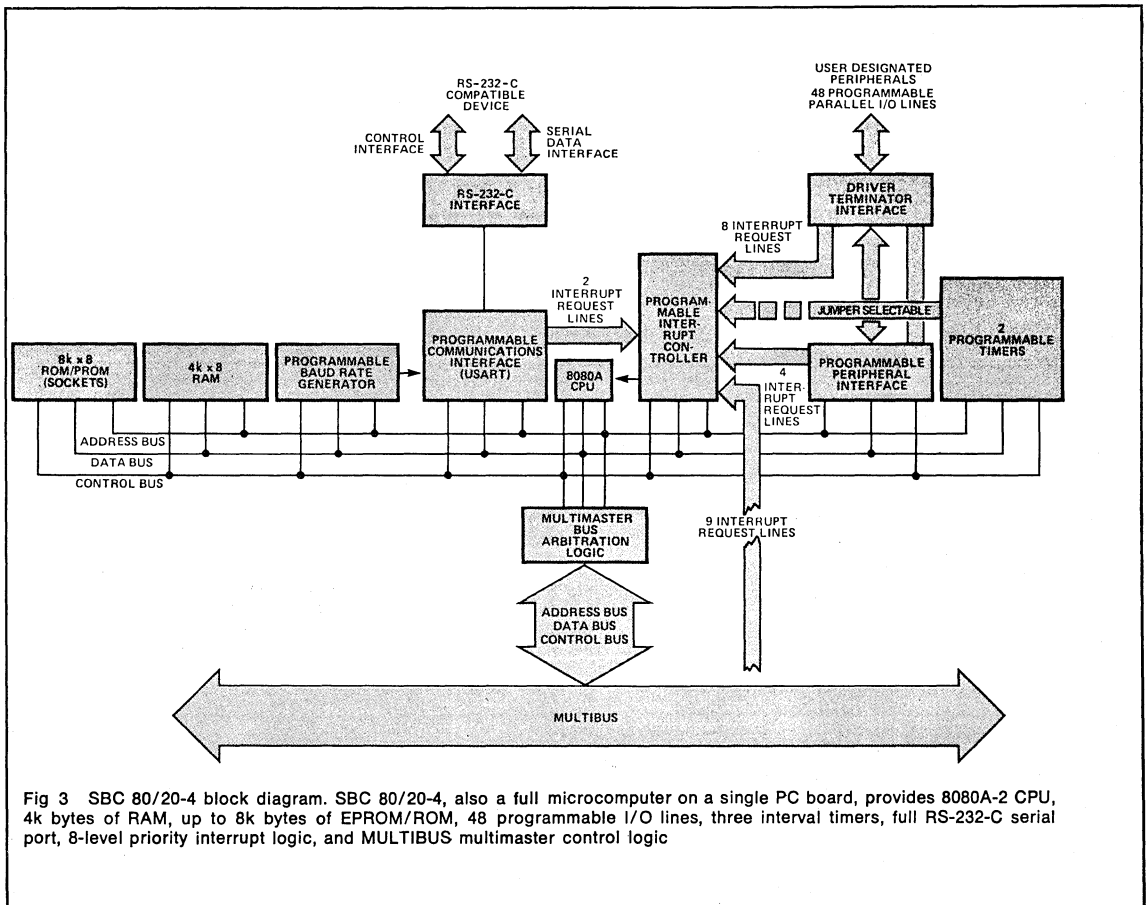
Fig 3 SBC 80/20-4 block diagram. SBC 80/20-4, also a full microcomputer on a single PC board, provides 8080A-2 CPU, 4k bytes of RAM, up to 8k bytes of EPROM/ROM, 48 programmable I/O lines, three interval timers, full RS-232-C serial port, 8-level priority interrupt logic, and MULTIBUS multimaster control logic

memory or I/O location, is a system bus request initiated. If no other master is currently utilizing the bus, this "new" master will be granted access immediately. However, this new master must wait if another master is currently utilizing the system bus. It continues to monitor the status of the system bus to determine when its current cycle may be completed. Thus, the MULTIBUS must provide a method for masters to determine whether or not another master is currently utilizing it.

Other masters may also simultaneously request the system bus. Arbitration must then be performed to resolve this multiple contention for the system bus. The MULTIBUS structure provides this arbitration in one of two techniques: serial (daisy chain) or parallel (encoded). The structure consists of four control lines that are synchronized by the common bus clock. These four control lines and the bus clock are active low. This is represented by the slash (/) character after each signal mnemonic. Control lines are as follows:

Bus Clock (BCLK/) —The negative edge of BCLK/ is used to synchronize bus arbitration. BCLK/ may be asynchronous to all CPU clocks, and it has a 100-ns minimum period. BCLK/ may be slowed, stopped, or single-stepped for debugging.

Bus Priority In Signal (BPRN/)—Indicates to a particular master that no higher priority master is requesting use of the system bus.

Bus Priority Out Signal (BPRO/)—Used with serial bus priority resolution scheme. BPRO/ is passed to BPRN/ input of master with next lower bus priority.

Bus Busy Signal (BUSY/)—Driven by bus master currently in control of MULTIBUS to indicate that bus is currently in use. BUSY/prevents all other masters from gaining control of bus.

Bus Request Signal (BREQ/)—Used with parallel bus priority network to indicate that a particular master requires use of the bus for one or more data transfers.

Serial (Daisy-Chain) Bus Arbitration

In a serially arbitrated MULTIBUS system (Fig 4) requests for system bus utilization are ordered by priority on the basis of bus location. Each master on the bus notifies the next lower priority master when it needs to use the bus for a data transfer, and it monitors the bus request status of the next higher priority master. Thus the masters pass bus requests along from one to the next in a daisy-chain fashion.

The highest priority master (Master 1) in the system will always receive access to the system bus when it requires it. There is no higher priority master to inhibit its bus requests, and its bus priority input line (BPRN/) is thus permanently enabled.

Masters operate asynchronously on the MULTIBUS. A master may thus be in the middle of a bus operation when a higher priority master requests the bus. Obviously, interruption of such an in-process cycle must not be allowed. The mechanism for avoiding such erroneous operation is the BUSY/ line. Upon being notified that access to the bus is possible, the master examines BUSY/. If this control line is inactive, the master will assert it, and complete its bus operation. If BUSY/ is already active, another master is currently using the bus. In this case, the master will examine BUSY/ upon every falling edge of BCLK/, typically once every 100 ns, until it becomes inactive. When BUSY/returns to its inactive state, the master will assert it, then complete its operation. The BUSY/line then inhibits higher priority masters from destroying a bus transfer cycle that may be already in progress.

The BUSY/ line is also controlled by a bus lock function on the SBC 80/05 and 80/20. This function allows a master, which currently has control of the bus, to retain control by independently asserting the BUSY/ line until it issues an unlock command that releases BUSY/. This permits a bus master to obtain exclusive control of the system bus for critical system functions,
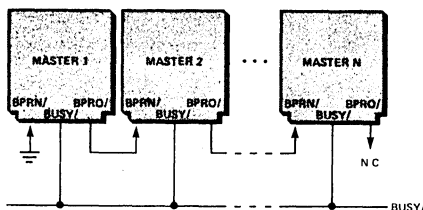
Fig 4 Serial bus arbitration. When any master requires use of MULTIBUS in serial (daisy-chain) priority mode, its BPRO/ line inhibits lower priority masters from system bus utilization. BUSY/ line is used to ensure that in-process operations of lower priority masters are not destroyed by asynchronous bus requests of higher priority masters

such as high speed memory or I/O data transfers and critical read-modify-write operations. With BUSY/ asserted in this way, all other masters will find the bus "in use" when they attempt to access it. Whereas system bus transfers normally take place on an interleaved basis (bus arbitration performed for each cycle), this bus lock function permits fast multiple-word transfers, when needed.

Two basic parameters determine the number of masters that can coexist on the system bus in serial bus arbitration mode. These are the BCLK/ cycle time and the BPRN/ to BPRO/ propagation delay of bus masters. Masters may be added to a system as long as the cumulative BPRN/ to BPRO/ propagation delay is such that the lowest priority master will always have its BPRN/ line driven inactive before the next BCLK/ falling edge after the highest priority master requests the bus. This worst-case timing condition is met as long as the following relationship is satisfied.

$$\sum_{i=1}^{N-1} (t_{BPRN-BPRO})_i < t_{BCLK} - t_{sh}$$

where

$(t_{BPRN-BPRO})_i$ = Propagation delay for master i
t_{BCLK} = Bus clock (BCLK) cycle time (period)
t_{sh} = Allowance for bus setup and hold times
N = Number of bus masters

Using serial bus arbitration and SBC 80 onboard clocks, up to three masters may coexist on the system bus. This number can easily be extended, if desired, by generating a BCLK with a longer cycle. The SBC 80/05 and 80/20 provide a jumper option which allows the onboard BCLK/ to be disabled. This allows the system designer to generate BCLK/ externally.

Parallel (Hardware-Encoded) Bus Arbitration

The parallel bus arbitration technique resolves system bus master priorities using external hardware. The parallel multimaster control line (BREQ/) comes into force in this case. Each master asserts BREQ/ when it requires access to the system bus. These lines are fed to a 2-chip parallel priority network. As with serial priority resolution, BPRN/ acts as the bus access enable input to each master. As Fig 5 illustrates, up to eight master priority levels are encoded by a 74148 priority encoder to a 3-bit code representing the highest priority master currently requesting the system bus. This code drives the 8205 3-to-8 decoder which asserts the proper BPRN/ line low to grant bus access to the highest priority master. The 74148/8205 propagation delay is less than 40 ns, easily fast enough to allow eight masters to coexist in this configuration utilizing a BCLK/ with a 100-ns period.

Systems requiring up to 16 masters may implement bus arbitration by utilizing two 74148 priority encoders and two 8205 decoders to provide a 16-level hardware priority network. The actual number of bus masters feasible on the system bus will also depend on bus drive/loading considerations. Even under this consideration, systems containing up to 16 masters are feasible.

Thus, single-board computer masters, in conjunction with the MULTIBUS control structure, provide off-the-shelf hardware solutions for the development of efficient multiple processor microcomputer systems. In addition to this hardware capability, the system designer needs to consider several software design issues.

Software Considerations

Several software operations, such as mutual exclusion, communication, and synchronization, are essential to proper multiple processor system operation. The MULTIBUS/SBC 80 functions that enable these software operations are examined.

Mutual Exclusion

In a multiple processor microcomputer system, there are
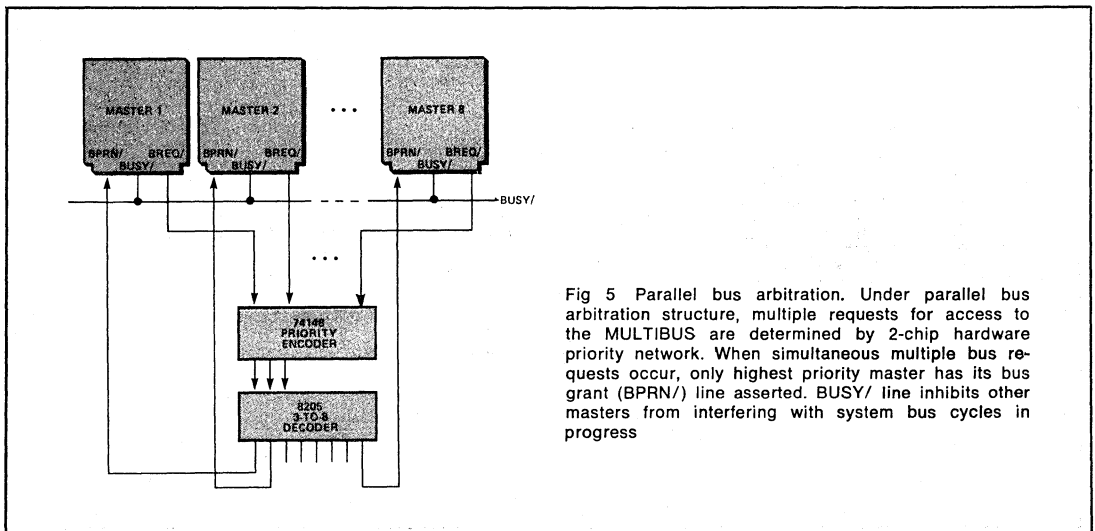
Fig 5 Parallel bus arbitration. Under parallel bus arbitration structure, multiple requests for access to the MULTIBUS are determined by 2-chip hardware priority network. When simultaneous multiple bus requests occur, only highest priority master has its bus grant (BPRN/) line asserted. BUSY/ line inhibits other masters from interfering with system bus cycles in progress

usually many resources that are shared by the processors. Such shared resources include common memory and peripherals. A properly functioning system must provide a mechanism to guarantee that asynchronous access to those resources is controlled in order to protect data from simultaneous change by two or more processors. Thus, some form of mutual exclusion must be provided to enable one processor to lock out access of a shared resource by other processors when it is in a critical section. A critical section is a code segment that once begun must complete execution before it, or another critical section that accesses the same shared resource, can be executed.

A Boolean variable can be used to indicate whether a processor is currently in a particular critical section (true) or not (false). Testing and setting this variable also presents a critical section. This function must be performed as a single indivisible operation; if it is not, two or more processors may test the variable simultaneously and then each set it, allowing them to enter the critical section at the same time. Such simultaneous entry would destroy the integrity of data and control parameters in global memory or cause erroneous double initialization of a global peripheral controller.

Mutual exclusion can be implemented as a software function alone, as described by Dijkstra[4], for n processors operating in parallel. The SBC 80/05 and 80/20 bus lock function mentioned earlier provides a means for using program control to simplify mutual exclusion. While the system bus is locked, the master can perform the indivisible test and set operation on the Boolean variable used to control access to a critical section without intervention from other masters.

Communication

Communication is an essential function that allows a program executing on one processor to send or receive data from a program executing on another processor. Typically, two processors communicate through buffer storage in common memory. One program, called a producer, adds data to buffer storage; another, called a consumer, removes information from buffer storage.

In a typical application, one master may produce buffers of data that are to be consumed by a program executing on another master that services an output device. Communication through buffer storage requires the operations of adding to and taking from buffers. These operations constitute critical sections that can be controlled by providing mutual exclusion around the buffer manipulation operations.

Synchronization

At times there is a need for one master to send a synchronization signal to another. In a sense, synchronization is a special case of communication during which no data is transferred. Rather, the act of signaling is used to "wake up" a program executing on another master. A program may "sleep," by waiting for a synchronizing signal, until it receives a wake-up signal that enables it to continue execution. Manipulation of synchronization signals requires mutual exclusion.
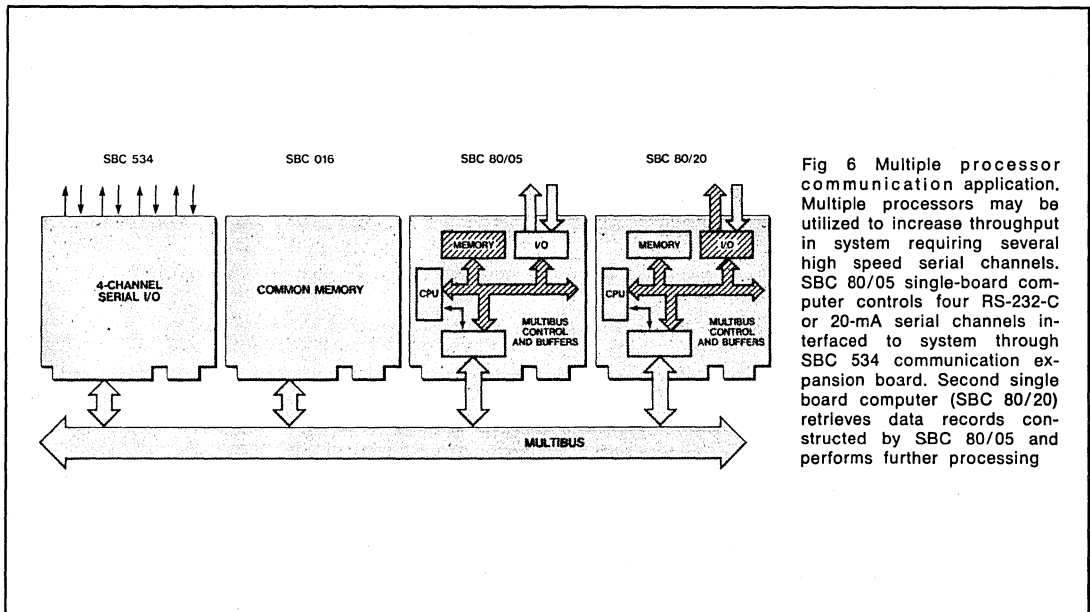
Fig 6 Multiple processor communication application. Multiple processors may be utilized to increase throughput in system requiring several high speed serial channels. SBC 80/05 single-board computer controls four RS-232-C or 20-mA serial channels interfaced to system through SBC 534 communication expansion board. Second single board computer (SBC 80/20) retrieves data records constructed by SBC 80/05 and performs further processing

System Initialization

In a microcomputer system that has multiple processors sharing a common system bus, a system initialization mechanism must be designed to set up the variables that control access to the shared resources. All single-board computers on the MULTIBUS begin execution simultaneously following a system reset. The bus lock function of the computers can be used by one specifically designated master to lock the bus immediately upon system reset and to perform system initialization for common resources before any other master attempts to access them. Since a locked bus has no effect on a single-board computer that is executing out of its local memory and using its local I/O, normal initialization by each processor can proceed while the shared resource initialization takes place.

Multiprocessor Applications

Two applications that are well suited to multiple processor microcomputer systems are examined. The first provides increased throughput, and the second allows shared resources.

Increased Throughput

Consider a system that is controlling multiple high speed serial communication channels in addition to other data processing activities. In this case, multiple processors may be utilized to increase system throughput. Such a system with four full-duplex serial channels operating at 4800 baud could produce interrupts every 250 μs. Interrupts at that frequency in a single master system would leave little time for other processing activities. In a multiple processor approach, one processor can be used to handle the interrupts from the serial channels, accumulate data into records, and then provide those records to another processor by placing them in common memory. The second processor is not burdened with the overhead of handling each character on an interrupt-driven basis, instead it is sent entire records of data available for further processing.

As shown in Fig 6, this application can be handled on the MULTIBUS with four boards. The SBC 80/05 single-board computer is used to service the communication board and prepare the data records. A 4-channel serial communication board (SBC 534) is used to provide the hardware interface for four serial communication channels. The SBC 80/20 single-board computer is used to process data records prepared by the SBC 80/05. Common memory is provided by the SBC 016 16k random-access memory (RAM).

Application of multiple processors to this problem requires communication through buffer storage. Two primitive operations, introduced by Dijkstra[4], can be used to simplify the communication and synchronization between the masters. These primitives, designated P and V, operate on non-negative integer variables called
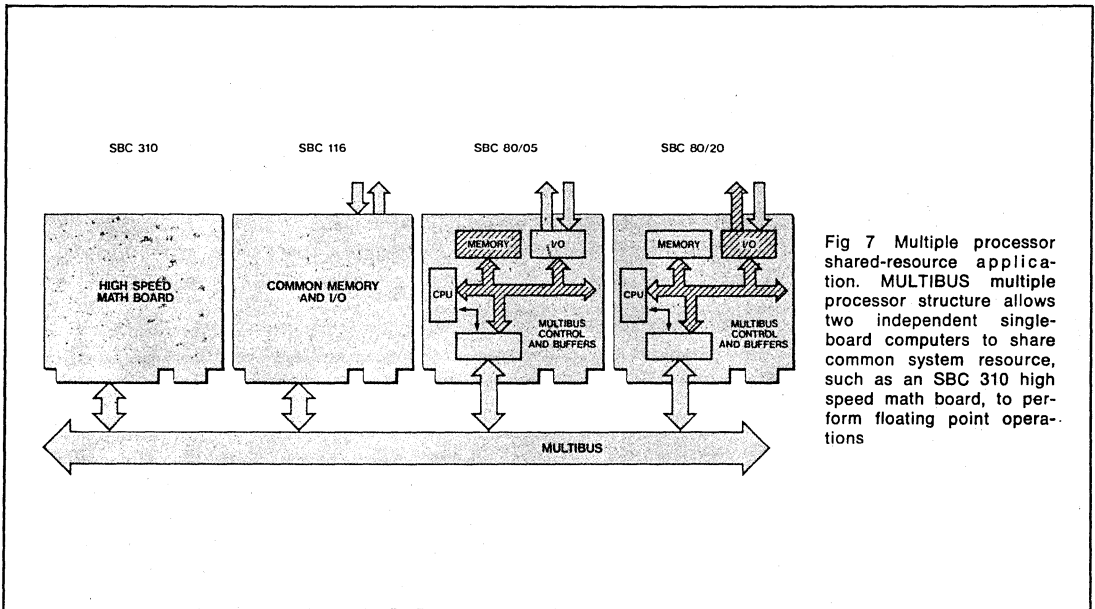
Fig 7 Multiple processor shared-resource application. MULTIBUS multiple processor structure allows two independent single-board computers to share common system resource, such as an SBC 310 high speed math board, to perform floating point operations

semaphores. The V procedure increments the semaphore (S) in a single indivisible operation. To make certain that fetch, increment, and store are not interrupted by another processor, the bus is locked during the operation.

Procedures for P and V primitive operations can be implemented in PL/M[6] as follows:

```
V:
  PROCEDURE (S$ADR);
  DECLARE S BASED S$ADR BYTE;
    OUTPUT(BUS$LOCK) = LOCK;        /* Lock MULTIBUS */
    S = S+1;                        /* Increment semaphore */
    OUTPUT(BUS$LOCK) = UNLOCK;      /* Unlock MULTIBUS */
END V;
```

The P procedure loops in a busy wait until S is greater than zero, at which time it decrements S. The act of fetching, testing, decrementing, and storing S is also an indivisible operation. Note that if several masters with different speeds are in a busy wait on the same semaphore, the solution presented may not be "fair" to the lower speed processor; that is, the lower speed processor would test the semaphore less frequently, resulting in an unfair advantage for higher speed processors.

Implementation of a procedure for the P primitive is shown in the following PL/M code.

```
P:
  PROCEDURE(S$ADR);
  DECLARE S BASED S$ADR BYTE;
    DO FOREVER;
      IF S > 0 THEN                     /* Test semaphore */
      DO;
        OUTPUT(BUS$LOCK) = LOCK;        /* Lock MULTIBUS */
        IF S > 0 THEN                   /* Retest semaphore */
        -DO;
          S = S-1;                      /* Decrement semaphore */
          OUTPUT(BUS$LOCK) = UNLOCK;    /* Unlock MULTIBUS */
          RETURN;                       /* Exit from P procedure */
        END;
        OUTPUT(BUS$LOCK) = UNLOCK;      /* Unlock MULTIBUS */
      END;                              /* and continue testing */
    END;
END P;
```

It is important to observe in the program listing that S is tested prior to issuing a bus lock. This initial test avoids continuous locking and unlocking of the system bus while looping in a busy wait. The second test is required because another processor could also have found S greater than zero and tried to enter the critical section at the same time.

With the P and V operations, semaphores can be used as resource counters in the buffer manipulation required for communication between the SBC 80/05 and 80/20. For example, a consumer program can use the P operation to decrement the number of full buffers and a V operation to increment the number of empty buffers. In a similar fashion, a producer program can use the P operation to decrement the number of empty buffers and a V operation to increment the number of full buffers. In addition to full and empty buffer counters, it is necessary to maintain linked lists pointing to actual full and empty buffers. A semaphore can be used to provide mutual exclusion around the manipulation of the linked lists. In the example that follows, three variables (FULL, EMPTY, and SEMA) are used to implement these functions. The two PL/M programs illustrate consumer and producer code segments, respectively. Note that the consumer performs initialization because it accesses the semaphores prior to the producer.

```
CONSUMER:
  DECLARE EMPTY BYTE EXTERNAL;         /* Number of empty buffers */
          FULL BYTE EXTERNAL;          /* Number of full buffers */
          SEMA BYTE EXTERNAL;          /* Binary semaphore */
  OUTPUT(BUS$LOCK) = LOCK;             /* Lock MULTIBUS */
  EMPTY = NUMB$BUFFERS;                /* Initialize semaphores */
  FULL = 0;
  SEMA = 1;
  OUTPUT(BUS$LOCK) = UNLOCK;           /* Unlock MULTIBUS */
  DO FOREVER;
    CALL P(FULL);                      /* Decrement full buffer */
    CALL P(SEMA);                      /*   semaphore */
                                       /* Decrement mutual exclusion */
    . . .                              /*   semaphore */
    . . .
    (Take data from buffer and
    place it in local memory,
    move buffer from full to
    empty linked list)
    . . . .
    . . .
    CALL V(SEMA);                      /* Increment mutual exclusion */
    CALL V(EMPTY);                     /*   semaphore */
                                       /* Increment empty buffer */
    . . .                              /*   semaphore */
    . . .
    (Process the data)
    . . .
    . . .
  END;
END CONSUMER;

PRODUCER:
  DECLARE (EMPTY, FULL, SEMA) BYTE EXTERNAL;
  DO FOREVER;
    . . .
    . . .
    (Prepare data in local
    memory)
    . . .
    . . .
    CALL P(EMPTY);                     /* Decrement empty buffer semaphore */
    CALL P(SEMA);                      /* Decrement mutual exclusion */
                                       /*   semaphore */
    . . .
    . . . .
    (Place data in a buffer,
    move buffer from empty
    to full linked list)
    . . .
    . . .
    CALL V(SEMA);                      /* Increment mutual exclusion */
    CALL V(FULL);                      /*   semaphore */
                                       /* Increment full buffer semaphore */
  END;
END PRODUCER;
```

Shared Resources

Another typical application for a multiple processor microcomputer system would be to allow sharing of a resource by two processors. For example, consider two independent processors that have a need for high speed mathematical functions. Although it may not be possible to justify a high speed math module for each system, such a module might be justified if it were to be shared by both processors. A multiple processor microcomputer system could provide the capability to allow both processors to share the math module and not interfere with their otherwise unrelated functions.

This application (illustrated in Fig 7) could be handled with four boards. The SBC 80/05 single-board computer is used to perform various data processing functions requiring high speed floating-point arithmetic. The SBC 80/20 single-board computer controls a process where high speed numeric computations are required. High speed floating-point mathematics functions for both single-board computers are performed by an SBC 310 high speed math unit. SBC 116 combination memory and I/O board provides 16k RAM, 8k electrically programmable read-only memory (EPROM), 48 parallel I/O lines, and an RS-232-C serial port.

The problem to be solved in this application is to ensure that only one processor has access to the shared math module resource at one time. Thus, mutual exclusion must be provided to control the access to the resource. The following PL/M function returns TRUE if access to a critical section, used to implement the mutual exclusion, has been granted.

```
ENTER$CRITICAL$SECTION:
    PROCEDURE (FLAG$ADR) BYTE;
    DECLARE FLAG BASED FLAG$ADR BYTE;
    DECLARE ACCESS BYTE;
        IF FLAG = BUSY THEN                 /* Test flag */
            RETURN FALSE;                   /* Return false if busy */
        ACCESS = FALSE;
        OUTPUT(BUS$LOCK) = LOCK;            /* Lock MULTIBUS */
        IF FLAG = NOT BUSY THEN             /* Retest flag */
        DO;
            FLAG = BUSY;                    /* Set flag busy */
            ACCESS = TRUE;                  /* and access TRUE */
        END;
        OUTPUT(BUS$LOCK) = UNLOCK;          /* Unlock MULTIBUS */
        RETURN ACCESS;                      /* Return either TRUE or */
                                            /* FALSE access */
END ENTER$CRITICAL$SECTION;
```

This PL/M function first tests the flag for the busy condition before issuing a busy lock. As in the P procedure described earlier, this initial test avoids continuous locking and unlocking of the MULTIBUS while a busy wait is being executed. The following procedure performs a busy wait operation on the flag used to control access to a critical section.

```
BUSY$WAIT:
    PROCEDURE (FLAG$ADR);
    DO WHILE NOT ENTER$CRITICAL$SECTION(FLAG$ADR);
    END;
END BUSY$WAIT;
```

Typical code segments illustrating the use of these procedures follow.

```
DECLARE MATH$BD$FLAG BOOLEAN EXTERNAL;    /* Flag must be */
                                          /* initialized */
    MATH$BD$FLAG = NOT BUSY;
    . . .
    . . .
    CALL BUSY$WAIT(.MATH$BD$FLAG);        /* Here we wait until */
    . . .                                 /* we have access */
    . . .
    (Process math functions)
    . . .
    . . .
    MATH$BD$FLAG = NOT BUSY;              /* Set flag not busy */
    . . .
    . . .
    /* We could also test and then do some other */
    /* processing if the math module is busy */
    IF ENTER$CRITICAL$SECTION(.MATH$BD$FLAG)
    THEN DO;
        . . .
        . . .
        (Process math functions)
        . . .
        . . .
        MATH$BD$FLAG = NOT BUSY;          /* Set flag not busy */
    END;
    ELSE DO;
        . . .
        . . .
        (Something else)
        . . .
        . . .
    END;
```

Conclusions

The motivations for implementing multiple processor microcomputer systems include enhanced performance and throughput. When the appropriate hardware/software design considerations are made, modularity is easily achieved. Hardware solutions to many problems are provided by means of a MULTIBUS structure and SBC 80 single-board computers that have multimaster capability. Through control of MULTIBUS functions, the software designer can perform multiple processor communication, synchronization, and mutual exclusion.

Even with these significant steps toward the simplification of multiple processor microcomputer systems, the design of such systems remains a complex software/hardware design task. The future trend of multiple processor microcomputer systems will be to simplify the software tasks of implementing communications, synchronization, and mutual exclusion. These functions could be performed in varying degrees by additional hardware bus functions.

Potential rewards for a multiple processor architecture include enhanced system throughput, improved real-time response, modular system expansion, and improved system reliability. These benefits will pressure the technology of parallel processing to include microcomputers in an increasing number of computer applications.

References

1. "SBC 80/05 Hardware Reference Manual," Pub 9800483, Intel Corp, Santa Clara, Calif, 1977
2. "SBC 80/20 Hardware Reference Manual," Pub 9800317, Intel Corp, Santa Clara, Calif, 1976
3. A. C. Shaw, *The Logical Design of Operating Systems*, Prentice-Hall, Englewood Cliffs, NJ, 1974, pp 59-78
4. E. W. Dijkstra, "Solution of a Problem in Concurrent Programming Control," *Communications of the ACM*, Sept 1965, p 569
5. "Intel MULTIBUS Interfacing," Pub AP-28, Intel Corp, Santa Clara, Calif, 1977
6. D. McCracken, *A Guide to PL/M Programming for Microcomputer Applications*, Addison-Wesley, Reading, Mass, 1978

intel®

September, 1978

Triple-bus architecture on a
single-board microcomputer

By Jim Johnson, Craig Kinnie and Mike Maerz
Electronic Design 15 / July 19, 1978

9800791A

Triple-bus architecture lets a single-board microcomputer's CPU operate at full speed while other system components share the main memory.
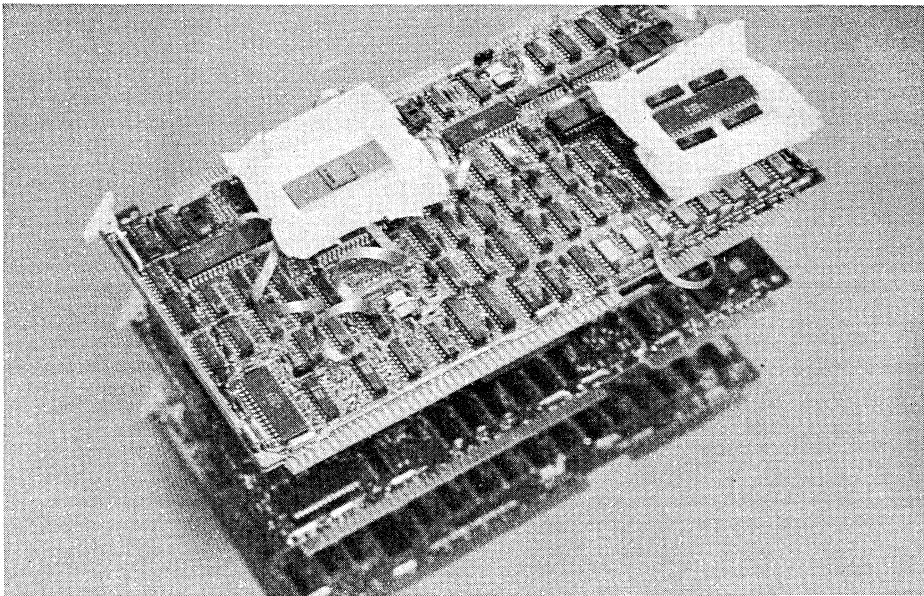
The introduction of Intel's iSBC 80/30 marks the beginning of the third generation of single board computer architecture. Two features separate the new microcomputer from second-generation single-board μCs. The major one is a triple-bus architecture that supports a dual-port memory. As a result, the on-board CPU does not tie up the main system bus (Intel's Multibus) when using the memory. Moreover, with two ports, the memory becomes a global resource, accessible via the three buses from the on-board 8085A CPU as well as from remote CPUs and other external devices in multimaster schemes.

In addition, the 80/30 contains two microprocessors: an 8085A acting as the master CPU and an 8041 single-chip microprocessor acting as a slave, or intelligent-I/O, processor.

Jim Johnson, Project Leader, **Craig Kinnie,** Project Manager, and **Mike Maerz,** Marketing Manager, Intel Corp., Santa Clara, CA 95051.

To appreciate the benefits of the 80/30's triple-bus, dual-port memory architecture, examine the following problem. Now that fully one fourth (16-kbytes) of the available memory space in a 64-kbyte μC system can reside on a single-board μC, the CPU must share these 16-kbytes with other system components, such as direct-memory-access devices, discs and other processors. What's the best solution—especially when, in many applications, 16-kbytes is all the memory that's required by the whole system?

Alternatives have problems

The most straightforward way is a split-bus architecture, in which both the CPU and the system have equal access to the memory (Fig. 1a). While the system bus will be able to handle memory access efficiently from devices tied to it, it will be tied up by the CPU—so external operations not related to memory accesses will be hindered.

SPLIT BUS:

SINGLE BUS:

DUAL BUS:

TRIPLE BUS:
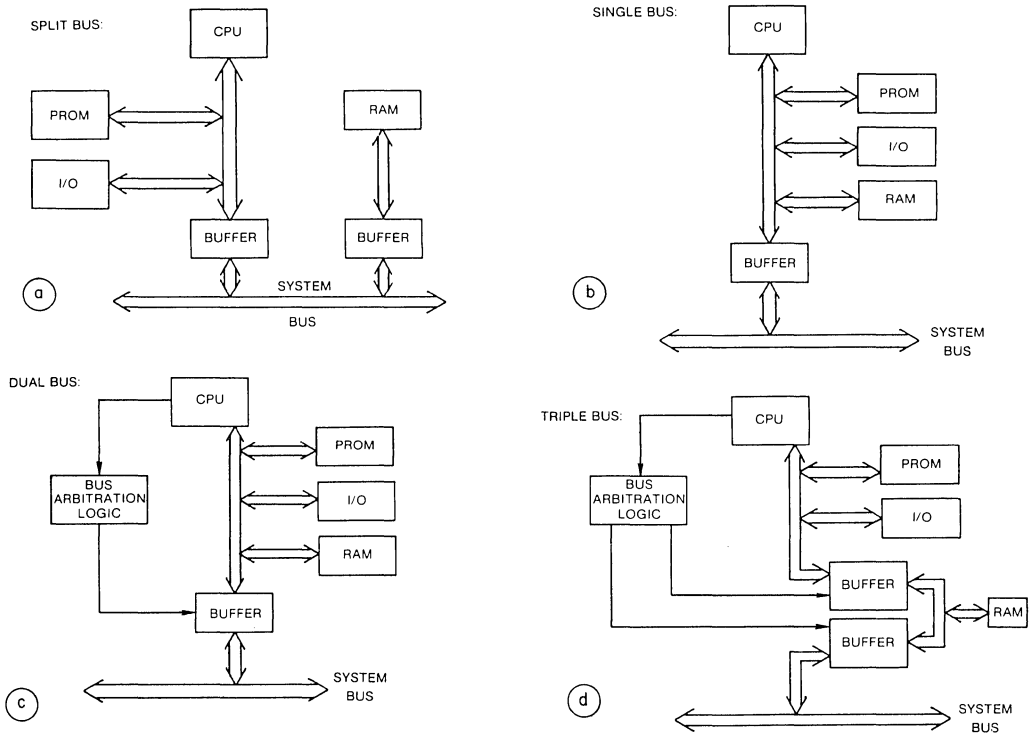
1. **Microcomputer-bus organizations** takes several forms: In a split-bus approach (a) the CPU and system have equal access to memory, but the CPU ties up the system bus; in a single-bus (b), the CPU encounters extra delays in using the system bus. A dual-bus structure (c) also has buffer delays, and no system access to on-board memory. But a triple-bus (d) avoids all these problems, allowing total system access to memory.

A single-bus approach (Fig. 1b) is hampered by buffer and bus-intervention delays which limit the CPU's performance. And dual-bus architecture (Fig. 1c), while granting the CPU exclusive access, does not allow other bus masters access to the memory. Also dual-bus suffers from buffer delays.

A triple-bus, dual-port architecture (Fig. 1d) provides the benefit of both single and dual-bus architectures: total system access and exclusive access by the CPU. But it also has its disadvantages: Dual-port architecture requires many buffers as well as access-arbitration logic. However, 20-pin octal buffers introduced by several manufacturers don't take up nearly as much board space or cost as much as equivalent standard buffers. Since the octal buffers come in unidirectional or bidirectional forms—and at nearly the same cost—the three-bus approach used on the 80/30 actually takes only as many packages as the split-bus approach.

Access arbitration is solved in the 80/30 with cycle status signals from the 8085A CPU. Instead of providing equal access to the RAM from both the CPU and the system, the arbitration logic is designed to favor the CPU. By assigning the default state of the arbiter

to the CPU, the logic anticipates a CPU memory access and reserves the memory until the cycle is complete.

In addition, if an on-board CPU access is imminent, a reservation signal derived from the 8085A CPU status signals, the ALE (address latch enable), the address, and the cycle status signals (SO, SI, IO/$\overline{M}$) will hold off bus contention. As a result, the CPU can operate at full speed without tying up the system bus.

Of course, this extra CPU performance cuts into the rest of the system's memory-access time. However, the penalty imposed by the arbiter is less than 200 ns—less than the time it would take a DMA device to regain control of the bus in the split-bus approach, where access must be interleaved.

A bus hierarchy

The three buses in the 80/30 hierarchy (Fig. 2) are an on-board bus, a dual-port (DP) bus and the Multibus (system bus). Innermost is the on-board bus, which connects the 8085A, all on-board I/O peripherals and ROM. The next bus in the hierarchy, the dual-port connects a dual-port controller, 16-kbytes of dynamic RAM and a dynamic RAM controller. The
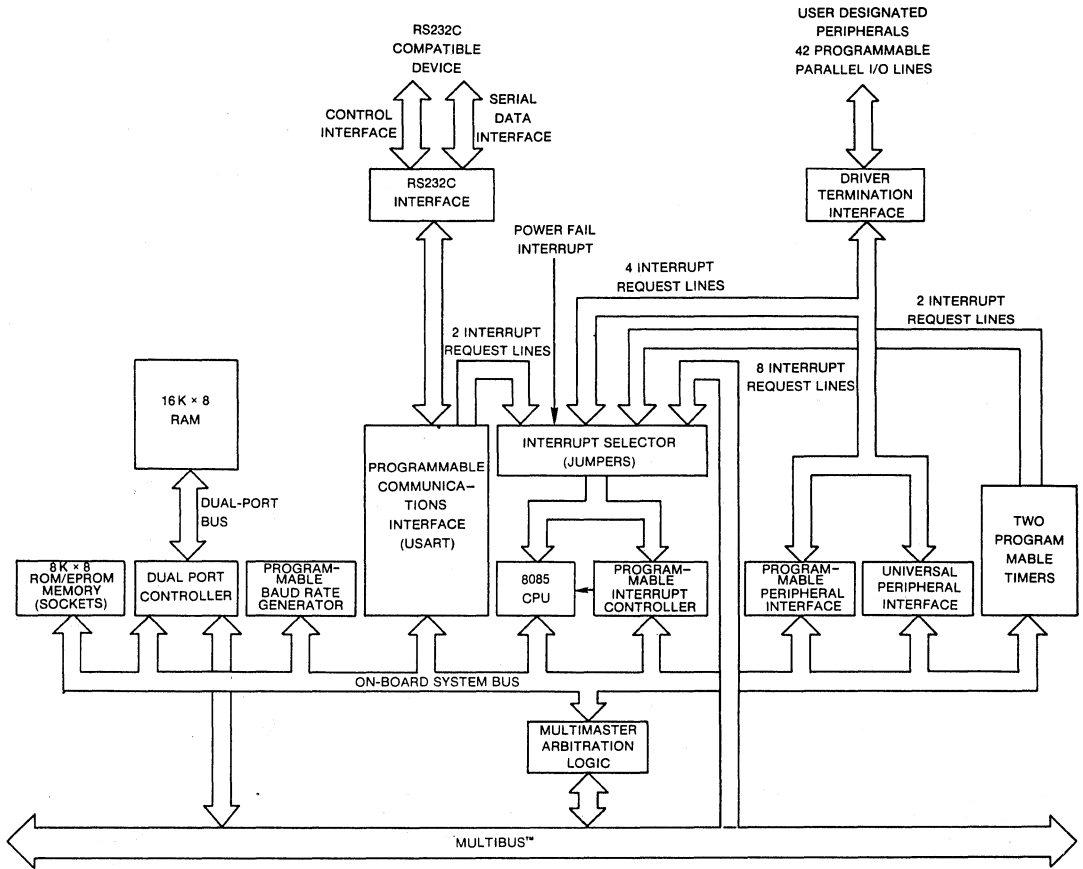
2. **The full 80/30 one-board microcomputer** is organized around its three buses: on board, dual-port, and the external-system Multibus. The main CPU, an 8085A, runs at 2.76 MHz, while an 8041A one-chip microprocessor serves as a peripheral controller or slave processor, running with a 2.6-ms cycle time.
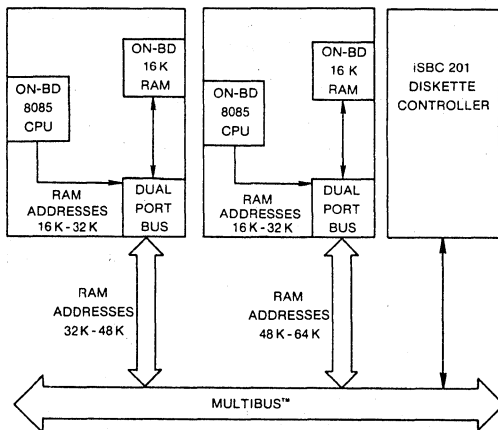
3. **The microcomputer's on-board memory** may be addressed independently by the on-board central processor and Multibus bus masters to increase the efficiency of usage of the total available memory space.

outermost bus, the Multibus, offers modules that permit either the expansion or addition of system resources.

With the on-board bus, the 8085A communicates with its on-board I/O and ROM (or PROM, if desirable) and the dual-port bus. Since the on-board bus permits access to the I/O and ROM only from the 8085A, all I/O and ROM (up to 8-kbytes are the 8085A's private property). And as a result, the 80/30 can operate on its on-board bus while another Multibus master uses the Multibus, accessing data from the board's dual-port RAM without reducing processor speed.

The dual-port (DP) bus contains 16-k of read/write memory, implemented with Intel's 2117 16-kbyte dynamic RAM and the 8202 dynamic RAM controller (DRC). The DRC interfaces the DP bus to the 16-kbytes of dynamic RAM, and provides an almost static-RAM type interface. It provides the system with multiplexed addresses, address strobes, and refresh control to the RAM, as well as refresh/access arbitration and acknowledges.

The RAM on this bus can be accessed from either

the 8085A on the 80/30 or the Multibus. The DP controller arbitrates the RAM requests and performs the bus exchanges.

The DP controller always leaves the DP bus under the control of the 8085A when it is not in use. This permits the 8085A to operate at maximum processor speed when controlling the bus, since there isn't any bus-exchange overhead. When the Multibus requests access to the DP RAM, the DP controller transfers control of the DP bus to the multibus, as soon as the DP bus is not busy. Once the Multibus transfer is complete, the DP bus is returned to the 8085A.

Multiple communication

The DP controller has two independent address decoders—one for decoding Multibus requests, the other for 80/30 requests. This not only permits the address space of the memory to be located in two different parts of memory (Fig. 3), it enables several 80/30s to talk to each other over the Multibus, while sharing the same on-board address as seen by the 8085A. Thus, one program can be loaded in any 80/30 without relinking and relocating the software for execution.

Each bus can communicate either within itself, or with the adjacent bus. Thus, the on-board bus cannot communicate directly with the multibus. However, when the CPU makes a bus request, the on-board and dual-port buses simultaneously determine if they can fulfill it. If the on-board bus can acknowledge the request, it does so, and the DP bus control is not required to determine if the DP bus can acknowledge the request. If the DP bus, not the on-board, can acknowledge the request, it does so, and the controller then lets the CPU use the bus. Thereafter, the RAM controller completes the operation and generates an acknowledge signal.

If neither the on-board nor DP bus can fill the bill, the Multibus is solicited by the CPU. Since a bus can only communicate with an adjacent bus, the on-board bus must request the DP bus to communicate with the Multibus via the DP controller. The on-board bus will retake control of the DP bus only after the request to use the Multibus is granted. This prevents lockout problems with the DP bus, where the CPU requests the Multibus when it is controlled by another bus master accessing the DP RAM.

How the 80/30 performs is directly related to how many buses it must use to complete a requested operation. The on-board bus always operates at maximum processor speed. The DP bus operates at maximum only if it hasn't been busy and a memory refresh cycle was not in process. The processor speed when the Multibus is used depends on bus overhead involved and the type of module requested.

The 80/30 boasts more than a three-bus architecture. For one thing, its I/O is designed to interface to a wide variety of external devices, including switches, motor drives, bistable sensors, displays,

The 80/30 in brief
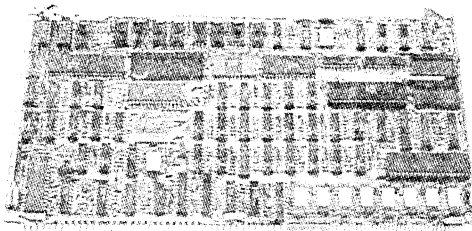
The iSBC 80/30 uses the latest LSI components to obtain the highest performance of any Intel single-board computer. Built on a 6.75 × 12-in. board, it contains the following features:
- 8085A central processor operating at 2.76 MHz.
- 16-kbytes of dual-port RAM using Intel's new 16-kbyte dynamic RAMs and 8202 dynamic RAM controller.
- Sockets for 2, 4 or 8-kbytes of ROM using Intel's 2758, 2708, 2716, or 2332 EPROMs or ROM replacements.
- A socket for Intel's 8041A/8741A universal peripheral interface (UPI) having 18 software-configurable I/O lines with sockets for drivers/terminators.
- A programmable serial-communication channel with RS-232 interface and programmable baud rate.
- Multibus control logic which allows up to 16 masters to share the system bus.
- 12 vectored priority interrupts.
- Two programmable 16-bit BCD or binary internal timers.

keyboards, printers, teletypewriters, communicator modems, cassettes and other computers. This versatility is provided with LSI programmable devices such as Intel's 8255 programmable parallel I/O device, 8251A programmable communication channel, 8253 interval timer, 8259 interrupt controller, and 8041A/8741A universal peripheral interface (UPI).

The slave processor

The ability to interface this wide variety of external devices is facilitated by the 8041A/8741A UPI (Fig. 4), which can be added to the 80/30. The UPI is a complete single-chip microcomputer which acts as a peripheral to the 8085A. It is completely user-programmable with 1-kbyte of ROM (8041A) or EPROM (8741A) memory for data storage. The UPI allows you to fully specify your control algorithm in the peripheral chip without relying on the 8085A. Devices such as printer controllers and keyboard scanners can be completely self-contained, relying on the 8085A only for data transfers.

The UPI is a powerful 8-bit CPU with a 2.6-ms cycle time and an instruction set optimized for bit manipu-
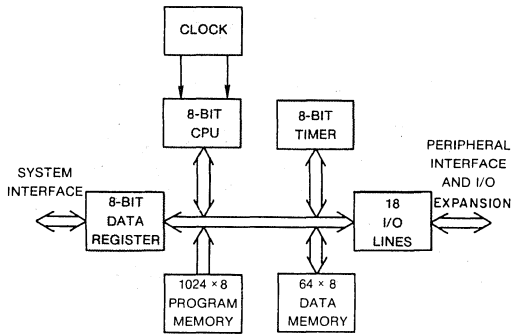
4. **The 8041A/8741A single-chip microcomputer** (UPI-41) has its own on-chip ROM and RAM and can be programmed to perform various peripheral control functions.
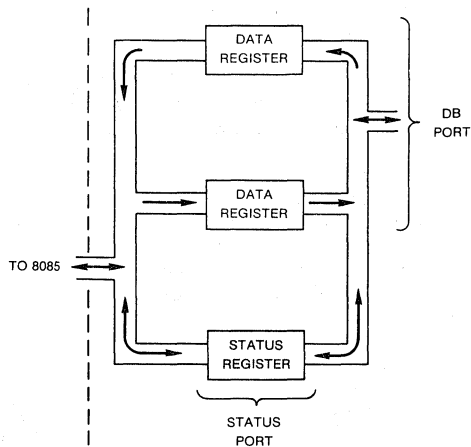
5. **The UPI's two data registers** are organized so that the 8085A CPU can write in just one register and read from the other. As a result, the two registers appear as one register to the main 8085A CPU.

lation and I/O operations. It contains an 8-bit counter/timer, buffers to communicate with the 8085A, and two 8-bit programmable I/O ports, which can be customized by software or by plugging in suitable line drivers or terminators into sockets. The UPI also has two input bits that it can test directly. An RS-232 driver and receiver on the 80/30 permit the UPI to be programmed as a simple serial-communication channel.

Interfacing to the on-board bus

The UPI interfaces asynchronously with the on-board bus using two data and two status registers. The UPI's two internal data registers appear to the 8085A as only one register, since one data register can be written into only by the UPI and read only by the 8085A, and the other can be written into only by the 8085A and read by the UPI (Fig. 5). This is done to prevent the two CPUs from simultaneously writing into a data register.

The UPI can communicate with the 8085A by loading a data register and then returning to its previous control task. The 8085A can periodically poll the UPI status port for the valid-read (VR) flag, which is set in hardware when the UPI writes to its data port, or the UPI can generate an interrupt to the 8085A via an I/O bit that can be programmed to be the VR flag.

Once the 8085A determines the VR flag is true, it can transfer the data to its own memory without disturbing the UPI. The VR flag is automatically cleared after the data are transferred. Similarly, when the 8085A transfers data to the UPI, a valid output (VO) flag is set and an interrupt to the UPI is generated (if enabled) automatically. Once the UPI transfers the data, the VO flag is cleared. The VO flag can also be programmed to a port bit for generating interrupts to the 8085A to indicate that the transfer is complete.

An extensive interrupt system

The 80/30 provides 12 vectored priority interrupts, four of which are handled directly by the 8085A's interrupt-processing capability and routed to fixed, unique memory locations. The remaining eight levels are handled via the 8259A programmable interrupt controller (PIC), which generates a unique memory address for each level. These addresses are equally spaced at intervals of four or eight (software-selectable) bytes. This 32 or 64-byte block may be located to begin at any 32 or 64-byte boundary in the 65,536-byte memory space. A single 8085A jump instruction at each of these addresses then provides the linkage to locate each interrupt-service routine independently anywhere in memory. The PIC provides a selection of four priority algorithms so that the manner in which real-time requests are processed may be configured to meet the requirements of the system under design.

The 80/30 also has two 8253-based programmable 16-bit BCD and binary timers/event counters, which can be used for a variety of functions. Both timers may be set to act as a rate generator (divide-by-N counter), a square-wave generator, a programmable retriggerable one-shot, or one of the timers can be jumper-selected as an event counter. In addition, an interrupt can be generated when a time interval has expired or when a specified number of events has occurred.

To see how useful the 80/30 can be, consider a supervisory control/monitoring system (Fig. 6) using an Intel iSBC 80/30 single-board computer, iSBC 201 diskette controller, and iSBC 732 analog input/output

```
                    ┌──────────┐
                    │   HOST   │
                    │ COMPUTER │
                    └──────────┘
                         ▲
                    SYNCHRONOUS
                    DATA
                    LINK

  ┌─────┐  RS232C      ╱ DUAL ╲    ANALOG    ANALOG
  │ CRT │  COMPATIBLE  │FLOPPY │  CONTROL   PROCESS
  └─────┘  DISPLAY     │DISKETTES│ SIGNALS  VARIABLES
           OUTPUTS      ╲      ╱
  DIGITAL                 SYSTEM
  CONTROL                 DATA
  INPUTS
  ┌──────────┐  ┌────────────┐  ┌─────────────┐
  │ iSBC 80/30│  │  iSBC 201  │  │  iSBC 732   │
  └──────────┘  │  DISKETTE   │  │ANALOG INPUT/│
                │ CONTROLLER  │  │   OUTPUT    │
                └────────────┘  └─────────────┘

  ◁═══════════════ MULTIBUS™ ═══════════════▷
```
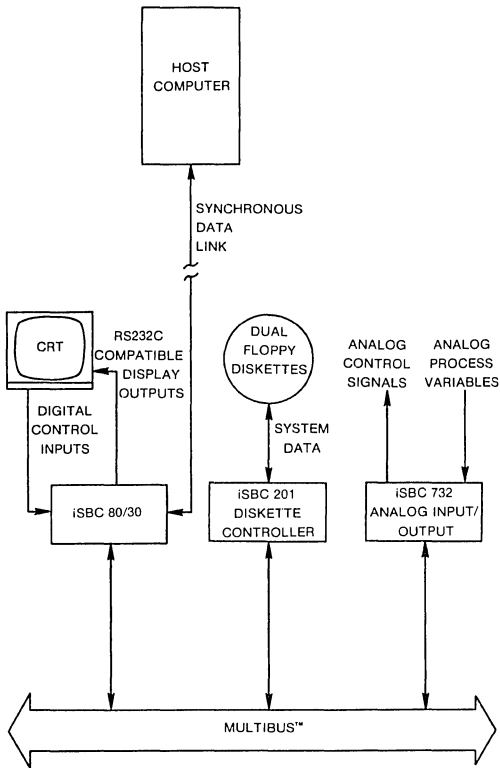
6. **In this application example,** the 80/30 forms the heart of a remote data-acquisition system. By taking advantage of the one-board microcomputer's dual-port memory and universal peripheral interface, the system achieves a combination of attractive cost and efficiency.

board. Here local commands and process-status signals are given and displayed on a CRT, which is interfaced via the iSBC 80/30 resident UPI and RS232C components. Process variables are converted from analog to digital using the analog I/O board. Control variables are passed over the Multibus from the 80/30 to the 732, where they are converted from digital to analog.

System data are logged on two diskettes, which are controlled by the 201. The controller board's on-board DMA interface accesses the 80/30's dual-port memory and stores the data on one of the floppy discs.

At the end of the day, a remote host processor, interfaced to the 80/30 via a modem (through the 80/30's 8251A and RC232C circuits) can request all or part of the diskette-resident data. Here, the 80/30 uses its on-board dual-port memory as a data buffer for transfers to the host.

Intel's RMX/80 real time executive, disc-file system and analog drivers provide the majority of the system's software.■■

Note: Multibus and iSBC are registered trademarks of Intel Corp.

Dual-port RAM Hikes Throughput in Input-output Controller Board

Craig Kinnie and Michael Maerz
Electronics

Dual-port RAM hikes throughput in input/output controller board

On-board random-access memory, accessible from system bus, makes input/output controller subsystem look like just another memory board to the host microprocessor

by Craig Kinnie and Michael Maerz, *Intel Corp., Santa Clara, Calif.*

☐ Input/output controllers based on microprocessors step up throughput in microcomputer systems by relieving the host processor of tedious, time-consuming control tasks—and a new design concept that increases the processing capability of this subsystem promises to hike throughput even more. It will cut the host intervention needed to transfer data and to run the controller.

In this configuration, all communications between the host processor and the controller are handled through a section of dual-port memory that resides in the controller subsystem. This setup allows more efficient transfer of large blocks of data from the I/O device to the system without contention over access to the system bus. It also simplifies interprocessor communications because the subsystem controller appears to the host processor simply as an additional RAM board.

Although this concept allows the subsystem to remain dedicated to its I/O control function and to assume a subservient role to the host processor, it has more processing power than previous generations of such controllers. Hence it has been dubbed the intelligent-slave concept by Intel, which applies it in the iSBC 544 intelligent communications controller.

The new subsystem architecture is divided into three major sections: dedicated I/O, dedicated computer, and dual-port memory (Fig. 1). The dedicated input/output,
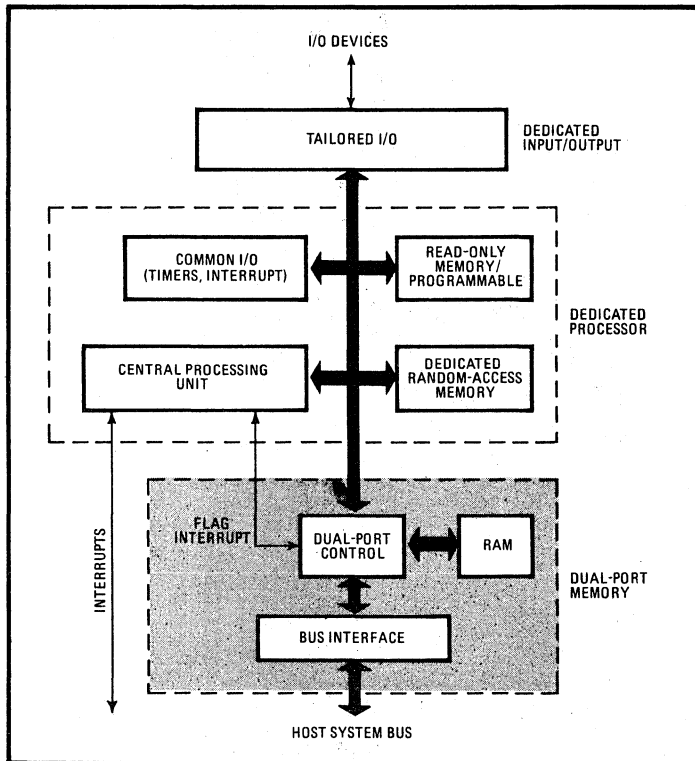
1. **Heart of memory.** New controller architecture includes the dedicated input/output circuitry and dedicated processor of an intelligent peripheral controller, but its heart is the dual-port random-access memory.
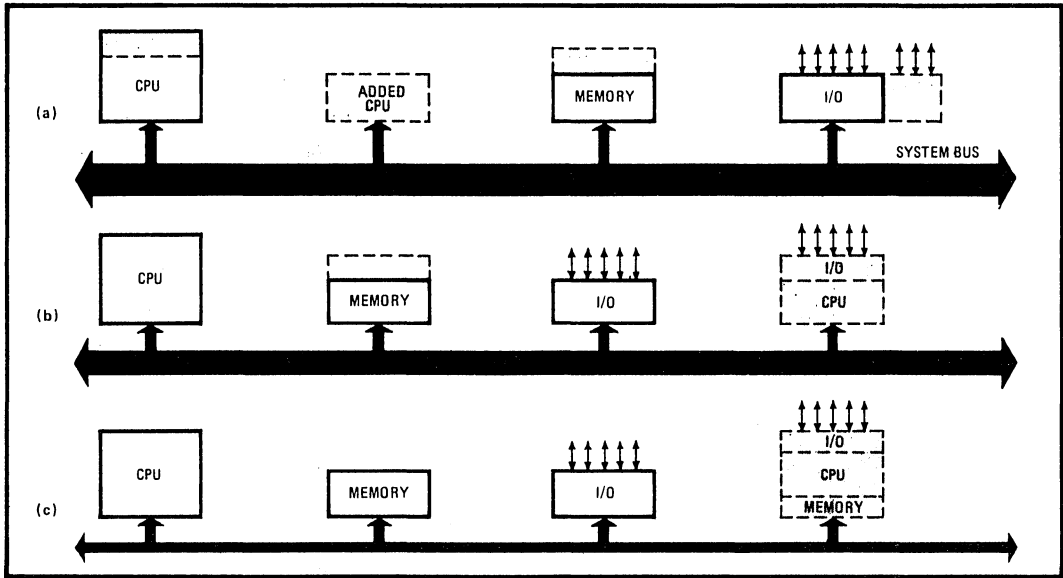
2. Performance advantages. In adding a real-time task to an existing real-time system, the load on the system bus is significantly reduced over the traditional multitasking approach (a) or the intelligent controller approach (b) by the intelligent-slave controller approach (c).
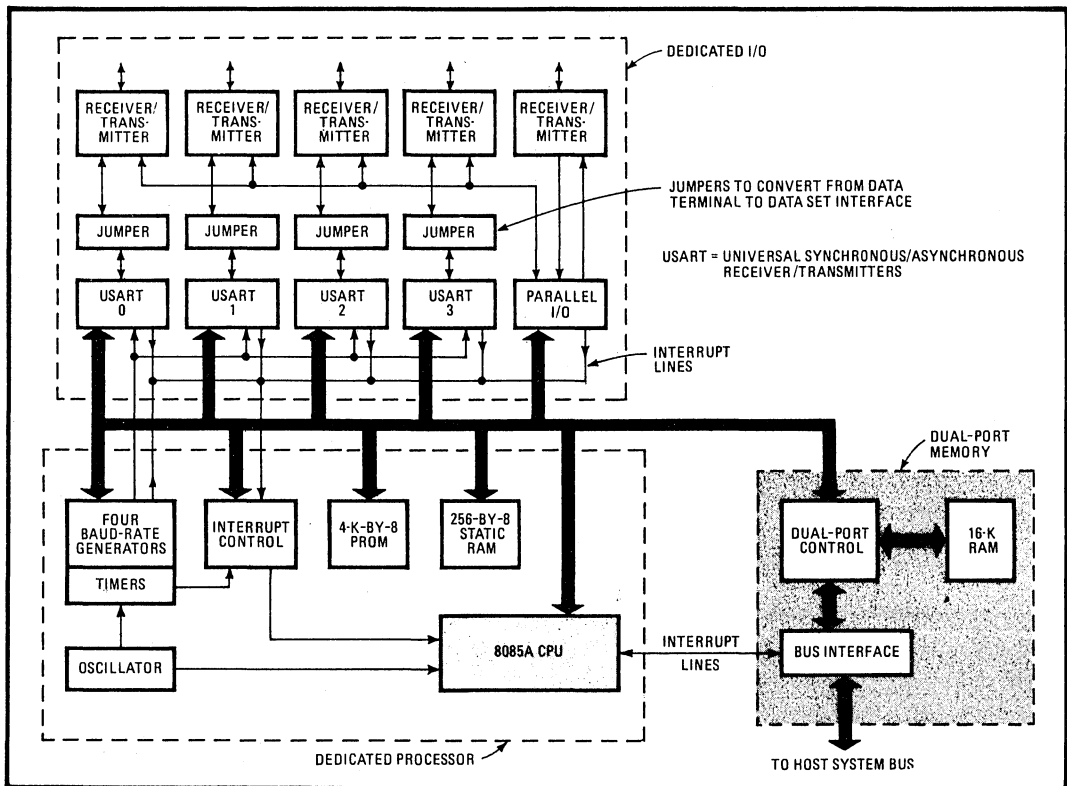
3. The 544. Based on the 8085A microprocessor with 4 kilobytes of PROM and 16 kilobytes of RAM, the subsystem is designed as a communications controller with four synchronous/asynchronous buffered serial I/O channels, and a 10-bit parallel I/O interface.

Dual-port RAM also shows up in new single-board computer

The concept of a dual-port read-write memory used in the iSBC-544 communications board is also employed in another new Intel product: its latest single-board computer, the iSBC-80/30. A dual port makes the 80/30's random-access memory directly accessible by the on-board 8085A central processing unit via internal busing without tying up the external system bus, the Multibus. At the same time, it also makes the RAM directly accessible by any other boards, like direct-memory-access controllers or other one-board computers that may be tied to the Multibus.

Moreover, the 80/30 adds its dual-port bus to the earlier iSBC computers' pair of buses: an internal bus, which hooks the CPU to peripheral chips and read-only-memory program storage and the system bus, over which the CPU and other boards communicated with RAM. Eight bits wide, the new bus is connected to a pair of buffer registers that coordinate, thus making the RAM accessible either by the internal bus or the system bus.

The objective is throughput: the CPU has priority in access to the on-board RAM. But since the access is not over the Multibus system bus, which might be tied up, there is no waiting. From the viewpoint of other system boards, the system bus is accessible a greater percentage of the time.

With the incorporation of 16 kilobytes of memory on the 80/30, Intel had little choice but to move to the dual-port, triple-bus architecture. The reason is that few system designs require more than 16 kilobytes, so in many applications all boards will be demanding access to the 80/30's memory over the Multibus. The CPU had better have priority to its RAM, through its own private line, lest the queue for the system bus bog down throughput.

The 80/30 also packs lots of extras, in addition to the total 16,384 bytes of read/write memory built with 2116 16-K dynamic RAMs. A pair of ROM sockets provide 4,096 bytes of program storage if fitted with 16-K erasable programmable read-only memories like the 2716. When pin-compatible 32-K erasable PROMs are available, program storage can be extended to 8 kilobytes.

Also on board is a socket for Intel's universal peripheral interface chip, the 8041 (or 8741 erasable-PROM version), which can function as a slave processor to drive peripheral devices. An 8251A universal synchronous/asynchronous receiver/transmitter is included for serial communications, and the 80/30 also boasts three 16-bit programmable timers. The 24 programmable input/output lines are brought out to sockets that accept quad line-drivers or -terminators for interfacing.

Ray Capece

consisting of the necessary peripheral chips, timers, buffers, and interface integrated circuits, tailors the controller to the application's I/O requirements.

The dedicated computer consists of a general-purpose microprocessor, electrically programmable read-only memory, dedicated RAM, timers, interrupt logic, and the decode and chip-select logic. The size and speed of the central-processing unit can be tailored to match the requirements of the dedicated I/O section.

The dual-port memory is the heart of the architecture and sets it apart from traditional approaches to intelligent controllers and multiprocessing. Passing all commands and data between the system and the controller's processor through this memory offers a number of significant advantages.

First, the dedicated computer's performance can be optimized for its applications. Its software always operates at full speed, since all required memory and I/O resources are immediately accessible on the board, without indeterminate delays caused by other system activity on the bus. This accessibility is especially important in real-time systems, since it allows the controller's performance to remain constant even though system bus activity may change.

Secondly, the architecture presents a consistent and convenient interface between the host CPU and all the controllers in the system, regardless of function. Because the controllers' dual-port RAM looks to the host CPU like just another location in system memory, the hardware and software problems associated with connecting multiple processors together are reduced to interfacing a number of identical intelligent memory locations.

Also, the architecture offers a degree of protection for the data in memory. The subsystem computer and soft-

ware can only alter that portion of system memory that resides in its own dual-port memory section. In contrast, traditional intelligent controllers have access to the entire system RAM and, should a malfunction occur, can destroy all of that memory.

System performance advantages

Because all processing assigned to the new controller's CPU takes place off the system bus, its architecture offers important performance advantages to the system. These advantages come from the appearance of the processed data blocks in system memory without consuming any system resources or bus time

The advantages of this approach are best demonstrated by comparing it to alternative means of adding a real-time task to an existing real-time system. In this case, the new task requires additional CPU, memory, and I/O resources.

The traditional multiprocessor approach of Fig. 2a expands CPU resources in one of two ways: software utilization of reserve capacity in the existing processor, or adding another processor. In either case, memory and I/O increments generally will be required.

The primary disadvantages of this approach are the increased complexity of the system software and the increased load on the system bus. Both will slow the existing real-time system unless it has been designed with adequate reserve. The system bus must also provide sufficient capacity for the incremental memory-execution and data-transfer operations. This additional bus load will also require that the primary real-time task can tolerate CPU delays due to bus contention.

The intelligent-controller approach of Fig. 2b has gained widespread use since the advent of the micropro-
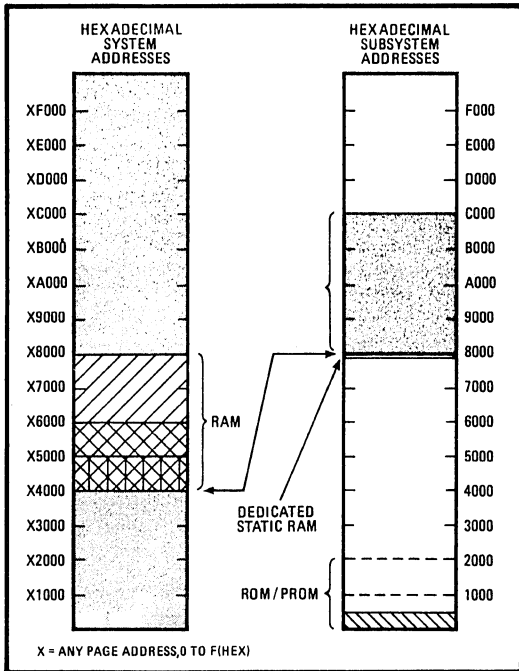
4. Memory mapping. The variable system memory addresses are always mapped into the on-board address of 8000HEX, providing software independence for the subsystem and the host.

TABLE: EIA RS 232-C SIGNALS PROVIDED AND SUPPORTED	
Carrier detect	Receive clock
Clear to send	Receive data
Data set ready	Ring indicator
Data terminal ready	Transmit clock
Request to send	Transmit data

cessor. This approach combines the CPU and I/O increments onto a single module that usually includes direct-memory-access transfer logic. In some cases the execution memory for the CPU is included.

This approach lessens the bus-loading problem since the I/O data transfers and some memory-execution cycles take place off the system bus. However, both CPUs' programs will have to tolerate delays caused by increased bus contention. Increased software sophistication is the primary disadvantage of this approach, much as with the multiprocessing approach of Fig. 2a.

The intelligent-slave approach of Fig. 2c can be viewed as a logical extension to the intelligent-controller approach. Combining the CPU, I/O, and memory increments creates a single module that has a minimal impact on the existing system software and bus loading. What's more, the subsystem can operate at full capacity without regard to other system activity. It can be programmed outside the primary system and then added with minimal impact on the system software or performance.

A limitation of the approach is the inability of the subsystem to transfer data into portions of the system memory space that reside off its board. This problem is minimized by the ability of the controller's RAM to serve as a substantial portion of the entire memory space addressable by the system. In this light, the on-board processor can be viewed as having a DMA capability limited to a portion of the system's address space.

In a system with more than one of the new controllers, the system CPU handles any data that must be trans-

ferred from one to another. Applications involving the transfer of large blocks of data would be best served by a central block-transfer device elsewhere on the bus.

The advantages offered by the new approach in this example of adding onto an existing system are just as applicable to a ground-up design. This modular approach to configuring real-time multiprocessing systems simplifies hardware and software design, as well as system integration.

While the primary design objective of the new architecture is operation in a multiprocessing system, it can provide significant utility as stand-alone processors. Thus these controllers incorporate a second mode of operation called the limited bus-master mode.

In this mode of operation the new controller can be used like a single-board computer as long as it is the system's only master of the bus. It can be connected to standard memory or I/O expansion boards to enhance its capability. It can even be used to drive other such controllers as long as they are used in the subsystem mode. This dual operational mode will allow the new controllers to serve a broad range of applications.

Communications first

Communications applications present complex processing requirements and an inherent real-time nature, so it is logical that a communications processor be the first of these new controllers to be marketed. The iSBC 544 intelligent communications controller can serve as a flexible front end to an iSBC system or as a cost-effective stand-alone processor configured as a terminal cluster or line concentrator. Its design (Fig. 3) incorporates an 8085A CPU, 16 kilobytes of dual-ported dynamic RAM, 4 kilobytes of PROM, programmable interrupt control, three interval timers, four programmable baud-rate generators, four synchronous/asynchronous buffered serial I/O channels, and a 10-bit parallel interface compatible with a Bell 801 automatic calling unit.

The dual-port memory block basically consists of the 16-kilobyte bank of random-access memory, which is accessible from either the system bus or the on-board processor through the dual-port controller. This memory block provides the primary means of communication between the system and the on-board 8085A. The port to the memory, which looks to the system bus like any other RAM card belonging to the system, features full 20-bit

addressing and a typical access time of 600 nanoseconds.

The interface's address-decode logic allows switching of the base address of the iSBC 544 to any 4-kilobyte boundary in the host system's address space. In addition, the user may reserve 8, 12, or 16 kilobits of the 544's memory for use by the on-board processor only. This reserved memory is not accessible from the system bus and does not occupy any system address space. The only restriction is that all of the unreserved memory reside in the same 64-K address page of the system memory.

This memory division can be a significant advantage in large 8-bit microcomputer systems. Only that portion of the controllers' memory needed to pass data between CPUs must be made accessible to the system. The remaining buffer and execution memory does not consume any system address space.

The net result is an increase in the system's overall memory capacity. For example, a microcomputer system that would usually be limited to 64 kilobytes of memory has a total memory capacity of over 190 kilobytes when driving seven 544s.

Address maps and interrupts

To the on-board processor, the base address of its memory is fixed at 8000HEX. Furthermore, all on-board addresses are fixed, so that multiple 544s operating on the same system bus can be running identical programs regardless of their base address on that bus. This capability necessitated the address-mapping logic to transform addresses from the system bus into the equivalent in the on-board address space starting at location 8000HEX (Fig. 4).

The address-mapping logic also implements the flag-interrupt feature. It provides an interrupt to the on-board processor whenever a byte is written into the 544's base address from the system bus, and a read from the on-board processor to the base address clears the interrupt. Since each 544 in a system has a different base address in that system's RAM, it also has a unique interrupt. This flag-interrupt capability is a key element in establishing a protocol for communications between the host CPU and the subsystems' processors.

The dual-port control logic is responsible for resolving contention over access to the memory and is designed to optimize the performance of the subsystem CPU. Unless the system bus has initiated a memory cycle before the on-board processor requests memory, that CPU runs at full speed. The maximum delay that can be encountered is one memory cycle. The arbitration logic actually reserves the memory for the on-board processor before it generates the necessary commands. This advance reserving guarantees that the on-board CPU will suffer minimum intervention from system bus accesses.

When the iSBC 544 is used in the stand-alone limited bus-master mode, the dual-port logic is disabled and the bus interface buffers are turned around to drive onto the bus. This reversal allows the on-board central-processing unit access to the memory of other subsystems or I/O expansion boards on the system bus.

The dedicated computer is built with an 8085A CPU operating at 2.76 megahertz, between 2 and 4 kilobytes of PROM and ROMs or 8 kilobytes of ROM using 2332
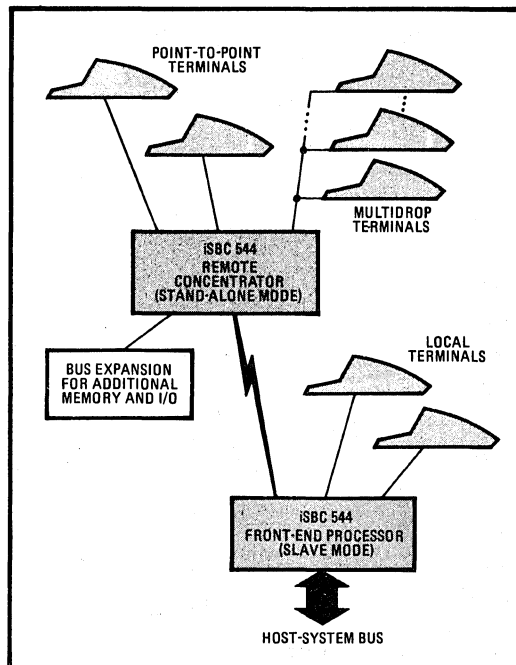
5. Communications applications. Two typical applications of the new iSBC 544 would be as a front-end communications processor to a microcomputer system and as a remote concentrator to a series of point-to-point or multidrop connected terminals.

mask-programmable parts, 256 bytes of static RAM, two 16-bit and one 14-bit interval timers, and a 8259 programmable interrupt controller for individual receive or transmit interrupt inputs for each serial port.

Special command-decode logic was added to the CPU to allow it to operate at maximum speed independent of other system activity. There are 21 sources of interrupt on the 544, including the separate transmit and receive interrupts for each port and separate timer interrupts. In addition to receiving an interrupt from the system, the 544 can also send an interrupt to the system bus via the 8085A's serial-output data line.

Since this controller is intended for communications applications, latched interrupts are provided directly to the CPU for loss of carrier and ring indicator for all four I/O ports. The ring-indicator and carrier-detect lines can also be monitored through the parallel port.

Dedicated I/O

The dedicated-I/O section of the 544 provides a high degree of flexibility and programmability. This results primarily from the inclusion of four 8251A universal synchronous/asynchronous receiver/transmitters. These devices are programmable for synchronous or asynchronous mode, character size, parity bits, stop bits, and baud rates. Data, clocks, and control lines are buffered with RS-232-C–compatible drivers and receivers to four 26-pin card-edge connectors. Each port is configured as a data-terminal interface, but may be converted to a
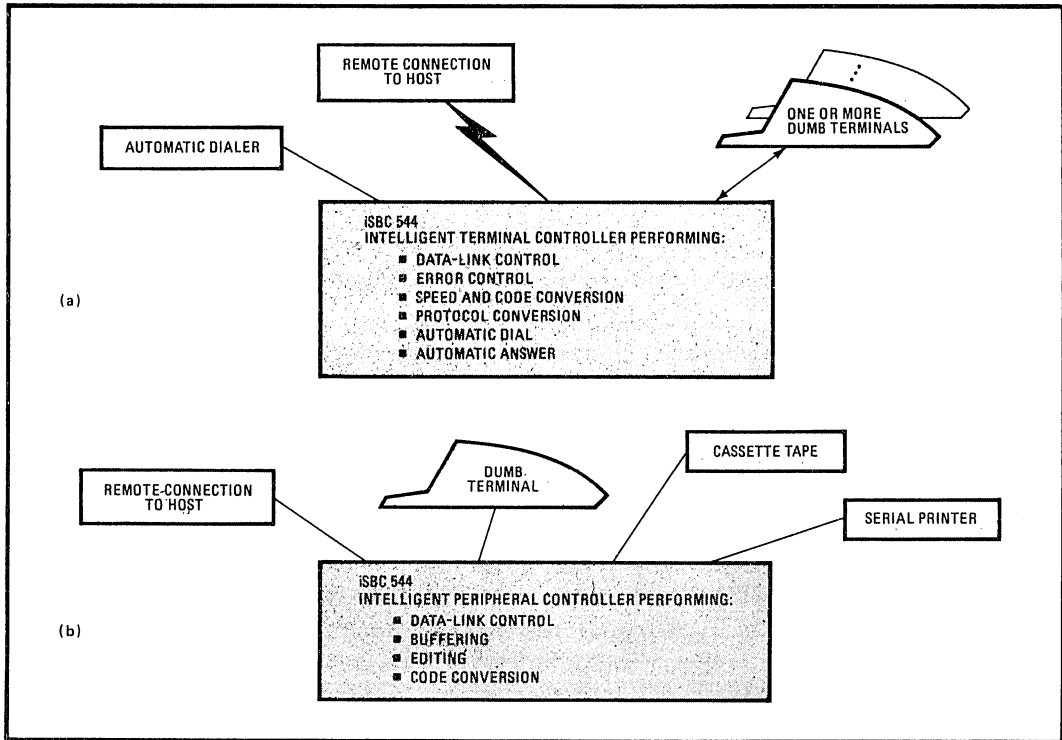
6. From slave to master. In its stand-alone mode, the 544 can operate as a bus master and be configured as an intelligent terminal controller connecting dumb terminals to a data link (a) or as a peripheral controller connecting RS-232-C–compatible units to the terminal (b).

data-set interface by changing a single jumper-plug assembly. The ports support most RS-232-C signals (those that are listed in the table).

A programmable baud-rate generator is also provided for each port. The range of baud rates available is 75 to 56 kilobits per second. The generators are implemented with 8253 programmable interval timers, which receive a jumper-selectable input frequency of 1.84 or 1.23 MHz. In addition, one of the CPU's interval timers can be converted to baud-rate operation and jumpered to any port to provide it with split-speed operation.

The 544 also provides a parallel port with four RS-232-C buffered input lines and six RS-232-C buffered output lines. This port is configured to interface to most automatic calling units but may be used as a general-purpose I/O port. It is implemented with an 8155 programmable peripheral interface that also provides the 256 bytes of static RAM and the 14-bit timer.

Applications

A likely common use of the 544 as a subsystem is as a front-end processor or terminal multiplexer (Fig. 5) in an iSBC system. The 544 performs all communications-related functions such as format control, code conversion, data-link control, error checking, data compression, and protocol management. It can handle multiple protocols, line speeds, and data formats.

All the system processor sees are the processed data blocks that appear in system memory. An automatic dialer could be added to provide a dial-up connection to a host processor or network.

Also shown in Fig. 5 is another 544 used in its limited bus-master mode as a remote concentrator and terminal controller. The line and memory capacity of the remote concentrator can be increased by the addition of standard iSBC memory and I/O expansion boards.

The intelligent-terminal controller shown in Fig. 6a is a prime example of a 544 used in the stand-alone mode. It can connect one or more dumb terminals to a data link and provide the necessary buffering, code conversion, and data-link control. It could also connect a terminal that happens to communicate in a different protocol to a new network or to more than one network.

The iSBC 544's multiple serial lines do not have to be used for communications. They can also be used to connect RS-232-C–compatible peripherals to the terminal (Fig. 6b). In this configuration, the 544 can provide message editing and formatting, bulk storage, and hard-copy output.

As this last application suggests, the 544 is the vanguard of a family of intelligent I/O controllers that will add tremendous increases in throughput and versatility to the iSBC line of single-board computers. The basic architecture will simplify the task of developing multiprocessing hardware and software solutions that will overcome throughput limitations. □

intel®

**ARTICLE
REPRINT**

AR-72

16-bit Single-board Computer
Maintains 8-bit Family Ties

By Robert Garrow, Jim Johnson
and Les Soltesz
Electronics

Technical articles

16-bit single-board computer maintains 8-bit family ties

Three-bus 8086-based board addresses a megabyte, communicates over expanded system bus

by Robert Garrow, Jim Johnson, and Les Soltesz, *Intel Corp., Santa Clara, Calif.*

☐ For the first time ever, 8- and 16-bit single-board computers can brainstorm over the same system bus. The iSBC 86/12 16-bit SBC has been designed to work intimately with its predecessors, the iSBC 80 family of 8-bit boards. What's in it for the user? Design flexibility—8-bit designs can be enhanced to 16 bits, developed software can be transported and, beyond that, 8- and 16-bit devices can be mixed in multiprocessing configurations. Several features make these options possible: a 16-bit CPU and instruction set designed for 8-bit compatibility; greatly expanded memory resources; and an extension of the Multibus specifications.

At the heart of the iSBC 86/12 is a 16-bit, high-performance metal-oxide-semiconductor 8086 central processing unit that operates at 5 megahertz. Because the 8086 instruction set is a superset to that of both the 8080A and 8085A 8-bit processors, the CPU can execute the full set of 8080A/8085A-type 8-bit instructions plus

a new set of 16-bit instructions. Thus, programs generated for 8-bit-CPU systems can easily be upgraded to run on the iSBC 86/12 using the software tools available with the Intellec microcomputer development system. Programs written in Intel's high-level programming language, PL/M, can be executed on both iSBC 80 and iSBC 86 products, preserving the software investment in 8-bit systems as a user moves into 16-bit applications.

Other features of the 8086 CPU are signed 8- and 16-bit arithmetic (including multiply and divide), efficient interruptible byte-string operations, and improved bit manipulation. Furthermore, the 8086 provides mechanisms for reentrant code, position-independent code, and dynamically relocatable programs.

This enhanced processing power is supported by the largest memory ever offered on a CPU board (Fig. 1). Memory address space has been extended over the iSBC 80 series to one million bytes. Up to 16 kilobytes of
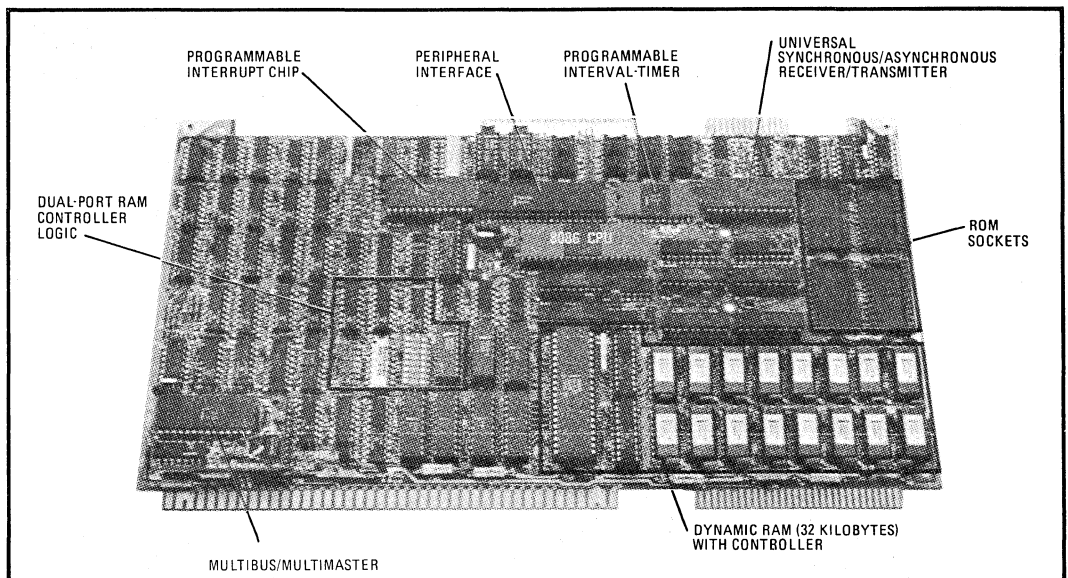
1. What a board. The iSBC 86/12 has 32 kilobytes of RAM and room for 16 kilobytes of ROM. The 5-MHz 8086 CPU executes 8080A/8085A-type as well as 16-bit instructions, including multiply and divide. Address space has been increased to a megabyte.
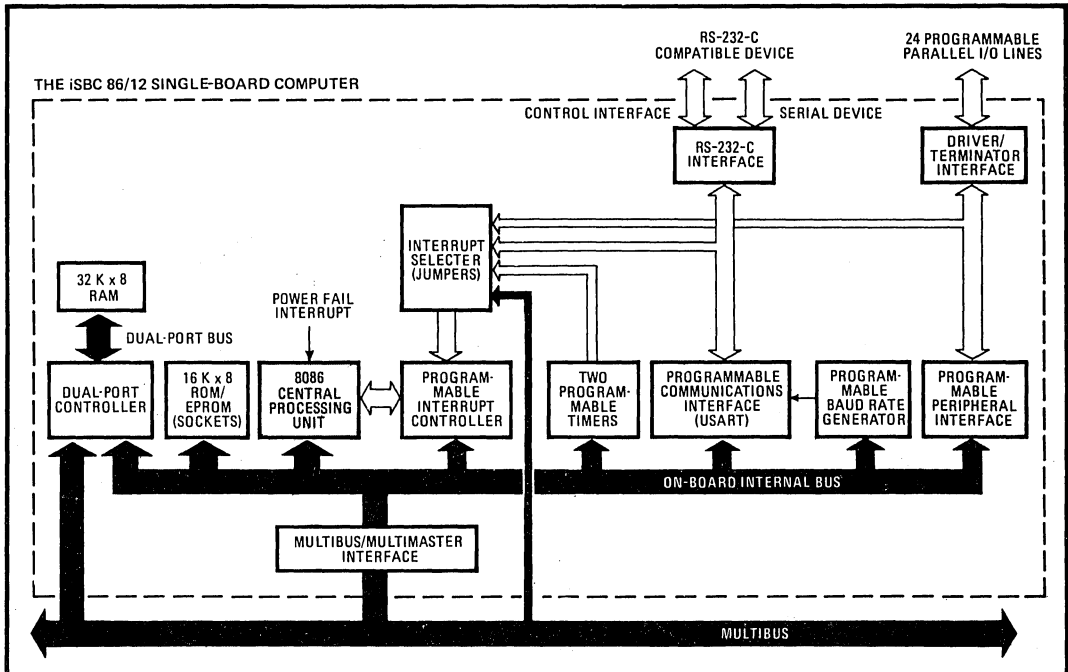
2. LSI + SBC = 86/12. A number of programmable LSI devices take credit for the power and flexibility of the iSBC 86/12. Note their interconnection to the three-bus hierarchy. When the 8086 requests a resource, the system bus is used only as a last resort.

read-only memory can be installed on the iSBC 86/12 itself. Furthermore, an additional 32 kilobytes of dynamic random-access memory with on-board refresh may be accessed independently by the CPU or by the system bus (Multibus).

Like the iSBC 80/30, the 86/12's RAM has dual ports to extend its use off board for access by other Multibus masters, including single-board computers, direct-memory-access devices, and peripheral controllers [*Electronics*, Aug. 17, p. 109]. All memory operations on the board occur independently of the Multibus, freeing it for external parallel operations. For applications that require data integrity at all times, a separate bus supplies power to the RAM and support logic via the edge connector. An auxiliary power source energizes the RAM in the event of power failure.

Multibus — the new look

To exploit the greater performance of the 8086 CPU and simultaneously make the iSBC 86/12 fully compatible with the iSBC 80 family of SBCs and expansion products, the Multibus specification has been extended to support 20 bits of address and 16 bits of data. The control lines, too, have been expanded to direct 8- and 16-bit data transfer over the system bus. These improvements enable the iSBC 86/12 to address directly a full one megabyte of system memory, access data in 8- or 16-bit word lengths, and recognize and acknowledge a variety of interrupts.

Address space has been enlarged to 1 megabyte by adding four address lines, A_{10}–A_{13}. Next, 8- and 16-bit

data operations have been defined to permit both types in the same system. This is done by reorganizing the memory modules, adding one new signal and redefining another. The memory is divided into two 8-bit data banks, which form a single 16-bit word. The banks are organized such that all even-byte-addressed data is in one bank (D_0–D_7) and all odd-byte–addressed data is in the other bank (D_8–D_F). A new bus-address signal has been defined to control the odd-byte bank called byte high enable (BHEN) during 16-bit operations. When active, BHEN enables the high byte of the data word from the addressed boards on the D_8–D_F Multibus data lines. A_0 controls the even byte bank and, when inactive, enables the low byte of the data word on the D_0–D_7 Multibus data lines. All word operations must occur on an even-byte-address boundary with BHEN active for maximum efficiency. (A_0 is inactive for all even addresses — see the table.) Word operations on odd-byte boundaries will be converted to 2-byte operations by the 8086, one for low-byte, one for high-byte. Byte operations can occur in one of two ways. The even bank is accessed when BHEN and A_0 are both low. This puts the data on D_0–D_7. To access the odd bank (normally placed on D_8–D_F during a word operation), a new data path has been defined. The active state of A_0 and the inactive state of BHEN are used to enable a swap-byte buffer, which places the odd data bank on D_0–D_7. This permits an 8-bit master access to both bytes of the data word while controlling only A_0. A_0 therefore specifies a unique byte and is not part of the word address, since all word operations are on even-byte boundaries.

Flexibility: LSI chips are the key

The iSBC 86/12 owes much of its flexibility to programmable large-scale integrated devices. An 8255A peripheral interface chip provides 24 programmable I/O lines that may be tailored to the customer's needs by simply programming the device for input, output, or bidirectional modes with or without handshaking abilities. In conjunction with the 8255A's configuration the user may then select appropriate line drivers and terminators for the I/O lines that can be inserted into sockets on the iSBC 86/12 board.

An 8251A universal synchronous/asynchronous receiver/transmitter is included to provide an RS-232-C interface for serial communication with other computers, RS-232-C-type peripherals (cassette tape, modems, etc.) or cathode-ray-tube terminals. The 8251A enables the user to customize the communication link. Synchronous/asyn-

chronous mode, data format, control character format, parity and baud rates from 75 to 38.4 kilobauds are all under program control.

For system timing functions an 8253 programmable interval timer provides two programmable timers, each of which may be used as a square-wave generator, retriggerable one-shot multivibrator or as an event counter.

The interrupt structure of the iSBC 86/12 encompasses nine levels with vectored priority. Eight of these levels are handled by an 8259A programmable interrupt controller chip, which may be configured for different priority processing modes in accordance with the application. One nonmaskable interrupt is available to immediately alert the CPU to catastrophes like a power failure, in which case the CPU can branch to an appropriate routine in memory to effect an orderly system shut-down.
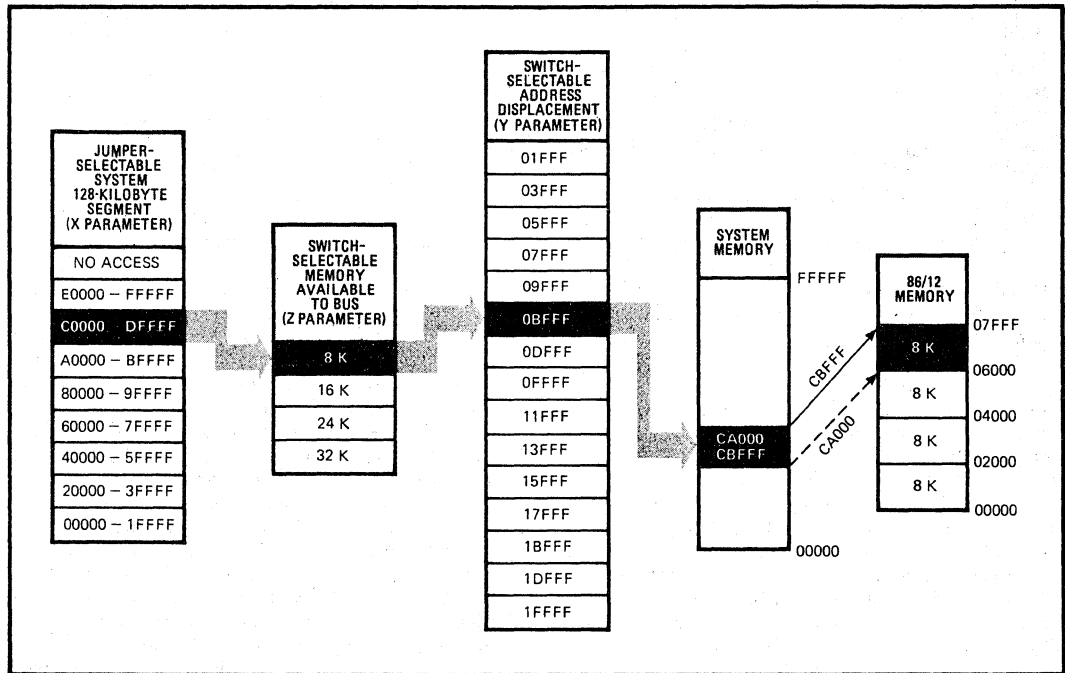
3. RAM, please. The 8086's view of on-board memory is fixed from zero to 07FFFH. When an outside master accesses this space, the DP controller performs the translation. Here, locations 06000H to 07FFFH are available to another master by addressing CA000H to CBFFFH.

Since all 8-bit accesses via Multibus are done on the lower byte of the data word, the iSBC 86/12 can access 8-bit memory or I/O devices from the system bus. This makes the iSBC 86/12 compatible with all iSBC 80 Multibus modules.

More interrupts, too

The iSBC 86/12 expands the previous Multibus definition of interrupts by creating two distinct types: non-bus-vectored (NBVI) and bus-vectored (BVI) interrupts. Each Multibus interrupt line can be individually defined

through software to be a BVI or NVBI. Using BVIs, the interrupt capability of a Multibus system can be increased to 64 bus-vectored-priority interrupts.

Using NBVIs, a slave module activates an interrupt line and the interrupted bus master generates its own restart address to service that interrupt. The Multibus address or data lines are not used. A BVI uses the Multibus address and data lines to communicate with the interrupting slave. When the slave module generates an interrupt, the bus master requires that module to generate the restart address. One additional command signal is
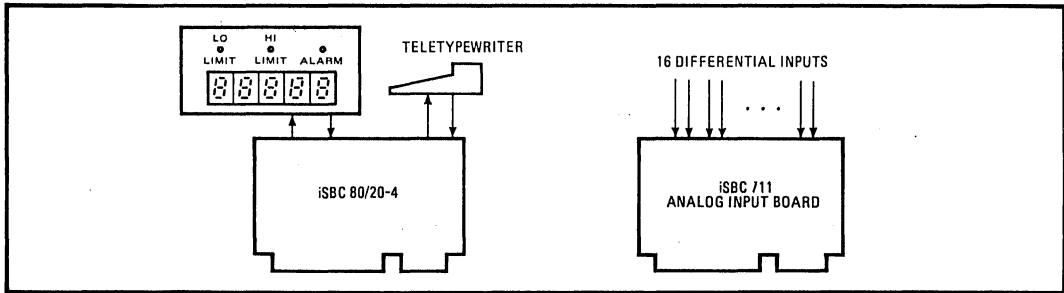
4. Open loop. Shown above is a simple alarm and monitoring system. The iSBC 711 analog-input board samples 16 differential inputs and the 8-bit iSBC 80/20 compares the inputs to the high and low limits. An alarm condition illuminates an LED and gets logged on a teletypewriter.
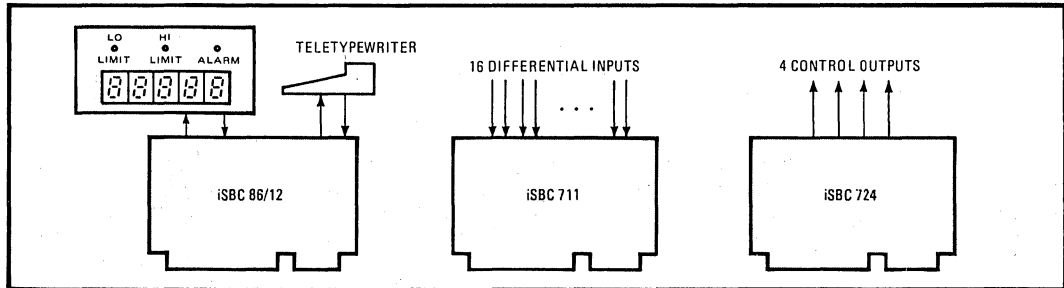
5. Closed loop. Suppose the system in Fig. 4 needs to be upgraded to handle a closed-loop system. For this application an iSBC 86/12 replaces the 80/20-04 to cope with the higher processing. The output control variables are handled by an iSBC 724 analog-output board.
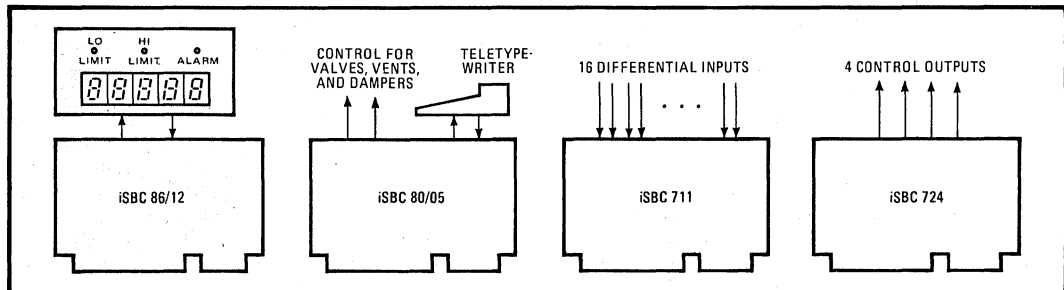
6. Multi/master. To enhance the control system in Fig. 5, add a dedicated CPU to control valves, vents, and dampers that, in turn, affect pressure and flow parameters in the system. This has been done by adding an iSBC 80/05 in a Multibus/multimaster arrangement.

defined—interrupt acknowledge (INTA)—to request the restart address from the slave module.

The iSBC 86/12 board architecture, like that of the 8-bit iSBC 80/30, is organized around a three-bus hierarchy: an on-board bus, a dual-port bus and a system bus (Multibus). All three buses have been expanded over their 80/30 counterparts to incorporate 20 address lines and 16 data lines.

The iSBC 86/12 architecture

The on-board bus links the 8086, all the I/O peripherals, and the read-only memory. Next in the hierarchy is the dual-port bus, which connects to the DP controller, 32 kilobytes of dynamic RAM, and the dynamic RAM controller. Finally, the system bus permits expansion of system resources through Multibus modules (Fig. 2).

The bus protocol of the iSBC 86/12 dictates that each

of the three buses communicate with an adjacent bus or operate independently. When the CPU makes a request for a resource, the on-board and dual-port buses simultaneously determine if their hardware can fulfill the request. If the on-board bus is able to acknowledge the request, it does so and the DP bus is not disturbed. (The DP bus is not interrupted to determine whether it can acknowledge the request.) The 8086 always controls the on-board bus, and requested operations can be completed without delay. If the DP bus is needed, it is requested and the dual-port controller grants the use of the bus to the processor. Thereafter, the dynamic-RAM controller completes the operation and generates an acknowledge.

If neither the on-board nor the DP bus can satisfy the request, the CPU asks for the system bus. The 8086 must use the on-board and dual-port buses to communicate
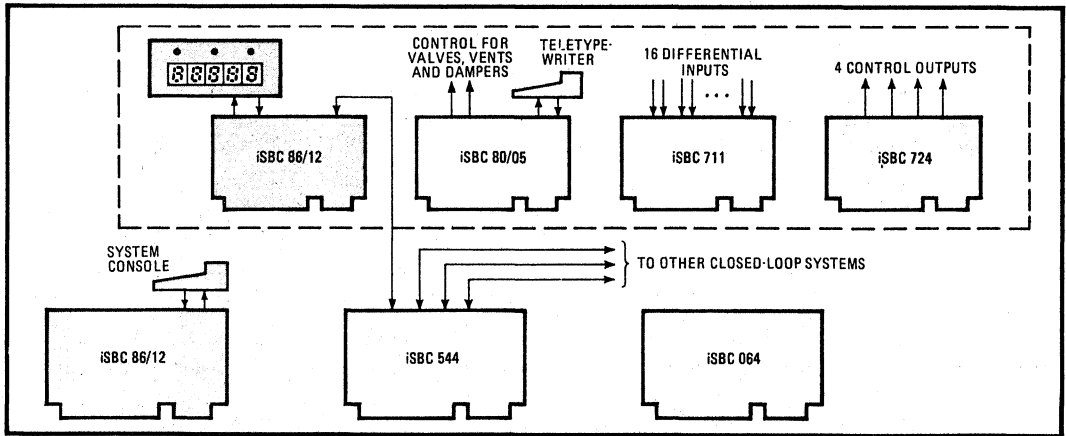
7. Four loops. An iSBC 86/12 can be used in conjunction with an iSBC 544 intelligent communications controller to perform a supervisory function for four closed-loop systems. The iSBC 544 controls the line protocol and the iSBC 86/12 processes the 544's data.

with the system bus. The 8086 takes control of the DP bus when the system bus is granted. This prevents lockout problems with the DP bus—that is, when the processor requests the system bus while another bus master has control of it and is accessing the dual-port RAM.

Naturally, the fewer the buses it has to access, the faster the iSBC 86/12 completes a transaction. The on-board bus always operates at maximum board speed. But the DP bus operates at maximum board speed only if it was not busy or taken up with a memory refresh cycle. When the system bus is brought into play, the processor speed depends on the overhead in acquiring it and the type of Multibus module being accessed.

With this three-bus architecture the iSBC 86/12 can be operating over its on-board bus at the same time as another Multibus master is using the system bus. It does so by accessing data from the DP RAM at no reduction in board speed. The on-board bus permits access only from the 8086. Thus all I/O and ROM are private to the 8086.

The dual-port controller has two independent address decoders—one for the 8086 and one for the Multibus. The 8086 decoder fixes the 8086's RAM addresses from hexidecimal 00000 to 07FFF using a fusible-link programmable ROM. The Multibus decoder allows the user to select any address range for the on-board RAM by specifying two parameters—a top-of-memory pointer and the size of the accessible memory. The TOM pointer (as seen by another Multibus master) can be set to any 8-kilobyte boundary in the 1-megabyte memory space. The amount of memory on the iSBC 86/12 accessible by another master can be set to 8, 16, 24, or 32 kilobytes (or no access) with an on-board jumper. For example, fixing the accessible memory size to 24 kilobytes provides the 8086 with 8 kilobytes of RAM that only it can access. This private area can be used for the processor's stack, interrupt jump table and other special system parameters that are generally protected from other Multibus masters. The only addressing restriction is that the memory block accessible to the Multibus cannot cross a 128-kilobyte boundary.

Suppose a Multibus master wants to load a program

into the iSBC 86/12's dual-port RAM for execution. Since the 8086's view of the DP-RAM address space is fixed, the Multibus address must be translated into the on-board 8086 memory space. The DP controller performs this translation by mapping the TOM pointer (as seen by other Multibus masters) to 8086-address 07FFFH, the top of the 8086's on-board RAM. Pointer − 1 is mapped to the top of 8086 on-board RAM − 1, and so on.

In the example shown in Fig. 3, the Multibus address selection is divided into three parts—two selecting the TOM pointer (X and Y) and one selecting the size of the accessible memory (Z). The TOM pointer is equal to a 128-kilobyte segment (X) plus address displacement (Y) from that segment. In this example, X is set to C0000H and Y is set to 0BFFFH, so the TOM pointer equals CBFFFH. Next, the size of the accessible memory (Z) is set, in this case to 8 kilobytes. This address translation makes the top 8 kilobytes of the 8086's RAM locations 06000H to 07FFFH available to another Multibus master when it addresses locations CA000H to CBFFFH. The 8086 still has 24 kilobytes (00000H to 05FFFH) of private memory.

Multiprocessing schemes

In multiprocessing systems, a master must be able to access the system without another master obtaining the bus. The iSBC 86/12 incorporates bus-arbitration logic to effect these transactions. Since the system bus is only requested when a system resource is needed, the iSBC 86/12 can perform true parallel processing with other iSBC 80 or 86 masters.

A typical example is the use of a common memory location that contains the status byte (busy/not busy) of a floppy-disk controller. When the floppy disk is needed, the master must first read the location and, if not busy, write the status word without another master obtaining the bus (to use the floppy disk). A bus-lock function on the iSBC 86, once enabled, allows the iSBC 86 to maintain control of the system bus until the lock is disabled by program control. This bus-lock function may

Data	Master	BHEN	ADR0	Data paths
8-bit even address	8-bit 16-bit mixed	0	0	LOW EVEN BYTES ← EVEN-BYTE BUFFER → D_{0-7}; SWAP-BYTE BUFFER; HIGH ODD BYTES — ODD-BYTE BUFFER — D_{8-F}
16-bit	16-bit	1	0	LOW EVEN BYTES ← EVEN-BYTE BUFFER → D_{0-7}; SWAP-BYTE BUFFER; HIGH ODD BYTES ← ODD-BYTE BUFFER → D_{8-F}
8-bit odd address	8-bit 16-bit mixed	0	1	LOW EVEN BYTES — EVEN-BYTE BUFFER → D_{0-7}; SWAP-BYTE BUFFER; HIGH ODD BYTES ← ODD-BYTE BUFFER — D_{8-F}

be activated in one of two ways—by an output bit from the resident 8255A peripheral-interface chip or by a software prefix on any 8086 instruction. The iSBC 86 can perform the test and set function by exchanging the accumulator with the memory location, preceding the instruction by a lock prefix. For example, the status word is read into the accumulator and, without another intervening bus cycle, a busy status is written. The accumulator is then tested: if busy—try again (writing a busy does not destroy status as it was already busy); if not busy, the floppy disk is now under the master's control and the status location is set to busy.

The iSBC 86/12: a design tool

For system debugging and full-speed execution, the iSBC 86/12 can be linked to the Intellec microcomputer development system. Programs generated on the Intellec system can be downloaded into the iSBC 86/12 RAM via cables. Through a virtual-terminal capability, the Intellec console can directly access an iSBC-resident monitor, which provides commands for software debug. Once the debugging cycle is completed, the user has the option of uploading the software back to the Intellec for storage on diskettes.

The Multibus and form-factor compatibility of the 8-bit iSBC 80 and 16-bit iSBC 86 single-board computers provide a degree of design flexibility previously unobtainable. Initial design problems can be solved with low-cost 8-bit hardware. As product requirements evolve, 16-bit performance can be added. Eventually, 8- and 16-bit multiprocessor solutions can be conveniently implemented.

Consider the application shown in Fig. 4, an alarm and monitoring system in a typical process plant. Sixteen differential inputs from pressure and flow transducers are sampled once every second, then compared to high and low limits previously entered through thumbwheel switches. The iSBC 711 analog-input board takes care of sampling the inputs and the 8-bit iSBC 80/20-4 compares the data to the high and low limits. Whenever these limits are exceeded, an alarm LED lights up and the alarm condition is logged on the system teletypewriter along with input identification, high limit, low limit and sampled value.

Closed loops

Instead of an open-loop system, suppose the design must be enhanced to control four output variables—thereby making it a closed-loop system. The sampling rate must be increased to once every third of a second and more processing will be required to run through the control algorithm and output the control-loop data. For this application, and iSBC 86/12 replaces the 80/20-4 to handle the higher processing requirements. An iSBC 724 analog-output board is also added to provide the four output-control variables (see Fig. 5). Carrying this example one step further, one may want to dedicate another processor to controlling valves, vents, and dampers that in turn affect pressure and flow parameters in the system. This can be done by adding an iSBC 80/05 in a multimaster arrangement as shown in Fig. 6.

Finally, an iSBC 86/12 can be used with an iSBC 544 intelligent communication-controller to supervise four closed-loop systems of the type shown in Fig. 6. The 86/12 of each system interfaces with the supervisory system via its serial interfaces, which are connected to the iSBC 544's serial ports (see Fig. 7). The iSBC 544 performs the control functions associated with the line protocol. The supervisory iSBC 86/12 can access the iSBC 544's dual-port memory and can perform further processing of the data received from the four closed-loop systems. In this configuration large amounts of memory may be required; since the iSBC 86/12 can address up to 1 megabyte, this presents no problem. □
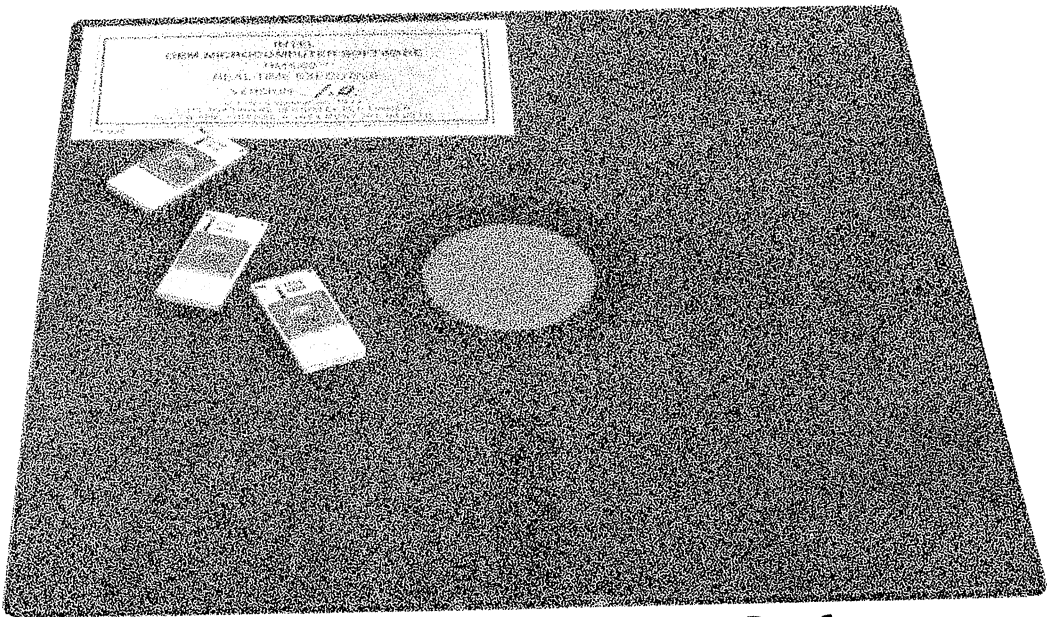
2 iSBC Software

iSBC SOFTWARE

INTRODUCTION

Software provided by Intel for the iSBC product family includes the RMX/80 Real-Time Multitasking Executive, the iSBC 801 RMX/80 FORTRAN Run-Time Package and the iSBC 802 Resident BASIC Interpreter.

The RMX/80 executive provides users of Intel's iSBC 80 series single board computers with a simple-to-use tool for implementing microcomputer software. Applications which monitor and control multiple asynchronously occurring events are prime candidates for use with the RMX/80 executive. The RMX/80 disk file system, terminal handler, analog driver and debugger have all been designed and optimized specifically for single board computer applications. RMX/80's modular design, consisting of a series of linkable and relocatable modules, can be configured to optimize system software for any specific application.

The iSBC 801 FORTRAN Run-Time Package and the iSBC 802 Resident BASIC Interpreter provide further enhancements to the RMX/80 executive. They provide users with a broad choice of programming languages for fast and efficient implementation of sophisticated system designs.

Publications reprinted in this section include an application note, a magazine article and a professional journal paper on the RMX/80 executive, and an application note on using FORTRAN with single board computers.

TABLE OF CONTENTS

October 1977

RMX/80™ Real-Time
Multitasking Executive

Thomas Rolander
OEM Microcomputer
Systems Applications

9800577A

RMX/80 Real-Time Multitasking Executive

Contents

INTRODUCTION

A large number of microcomputer applications require the ability to respond to events in real time. RMX/80 provides the system software around which you can build a real-time multitasking application on Intel SBC 80 Single Board Computers. In addition, RMX/80 increases the utilization of a Single Board Computer by allowing its resources to be shared among several tasks executing concurrently. Synchronization of these multiple real-time tasks is handled by RMX/80, freeing you to concentrate your major programming efforts on your application.

This application note begins with an overview of RMX/80. Readers who are familiar with the material presented in the RMX/80 User's Guide may choose to skip to the next section, a description of how to use RMX/80 and the steps involved in using it by describing two applications.

- An interrupt driven minimal terminal handler for a CRT or Teletypewriter.

- A closed-loop analog control subsystem utilizing the Intel SBC 711 analog-to-digital board.

Each example has diagrams illustrating the relationships between its tasks and exchanges. These are useful tools in conceptualizing the activities taking place in real time. Program listings of the applications are interspersed with text describing the application.

OVERVIEW

Real-time systems provide the ability to control and respond to events occurring asynchronously in the physical world. Later in this application note, a process control application is described that monitors and controls the temperature within several chambers. The system controls the process by simply turning on and off a heat source. The system could also display the temperature on an operator's console and permit entry of new setpoint temperatures and error ranges.

A single large program could have been used to perform the functions in a sequential manner. However, this approach may not permit an operator to enter control variables at the same time the process is being monitored and controlled. In contrast, real-time systems do not operate sequentially. A number of events may all be happening at the same time. This concurrence of events is a distinguishing characteristic of real-time systems.

BASIC CONCEPTS

There are basically three concepts that the user must master to effectively use RMX/80. The first is the task, an independent program which competes for resources within the system. The second concept is the message. Messages convey data and synchronization information between tasks. The third concept is the exchange. An exchange enables one task to send a message to another. As we will see later, the interaction between tasks and exchanges enables the user to implement mutual exclusion, communication, and synchronization. Mutual exclusion is a technique that controls access to a shared resource such as an I/O device or a data structure.

Task

Under RMX/80, the user codes a separate program, known as a task, for each event. An arbitrary number of these tasks execute concurrently and are subject to synchronization as required by their functions. Tasks share resources such as data structures and can communicate between themselves.

Message

A message is a unit of communication between tasks. Together with the exchange mechanism, it conveys information between tasks and can synchronize their operations.

Exchange

RMX/80 uses message exchanges for task-to-task communication. An exchange is a pair of queues represented by a data structure at which messages are left by one task to be picked up by another. Tasks may send messages to an exchange, and may wait for messages at an exchange. A task which waits for a message may perform a timed or an untimed wait. A timed wait will terminate upon the receipt of a message or at the end of the specified period of time, even if it has not received a message. When a task does an untimed wait for a message it is guaranteed that the task will not execute again until a message is available for it. A representation of the exchange data structure is shown in Figure 1.

GENERAL CHARACTERISTICS

In addition to the basic concepts of tasks and exchanges, several other general characteristics of RMX/80 are relevant in this overview.
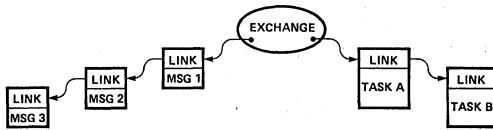
Figure 1. Exchange Data Structure

System Time Unit

RMX/80 uses a system time unit that is the period of time between "ticks" of the system clock. The standard RMX/80 system time unit is 50 milliseconds. The system time unit provides timing and user task scheduling. A task may wait at an exchange for a specified number of system time units and then continue execution. A task could be written to generate messages at specific time intervals. Tasks waiting for the messages would then be scheduled according to those time intervals.

Message Producing/Consuming Tasks

In general, tasks can be classified as message producing or message consuming tasks. The processing flow of these types of tasks are usually cyclic in nature and can be shown as follows.
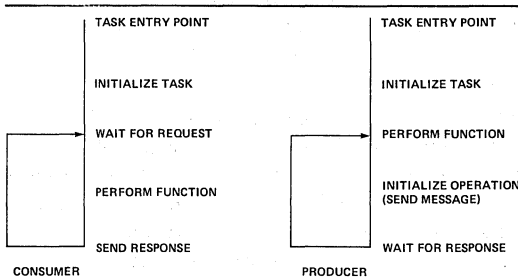
Figure 2. Message Producing/Consuming Tasks

A consumer task waits for a message to be posted at a particular exchange and takes control of the processor only when it has received a message and no other tasks of higher priority are ready to execute. The consumer task performs some action based upon the message and then simply resumes waiting until the next message is received. Usually, the consumer task acknowledges completion of its function by sending a response message to some other exchange associated with a task.

A producer task initiates its function by sending a message to another exchange and then surrenders control of the processor. The task continues to wait until it receives a response to its message.

Notice that the distinction between these types of tasks is relative since most tasks both produce and consume messages. However, the producer/consumer concept helps clarify the general structure of tasks—tasks are typically programmed loops. A producer task performs a function, sends a message, waits for a response, then loops back to begin again. A consumer task waits for a message, performs a function, sends a response, then loops back to wait again.

Interrupts

Hardware interrupts are treated as messages from peripheral devices for which a task can wait, as if the interrupt were a message from some other task. These messages arrive at particular exchanges, called interrupt exchanges, but are otherwise treated as described above. The system provides the ability to mask particular interrupts so that no messages ever arrive at a particular interrupt exchange associated with the masked interrupt. In the event that the overhead associated with turning an interrupt into a message is too high, the interrupt can be treated by the user directly via a user supplied interrupt service routine.

Task States

Tasks may exist in a number of states. A task is *running* if it actually has the processor executing instructions on its behalf. A *ready* task is one that could be running (any wait for a message or time period has been satisfied), but a higher priority task is currently running. A task is *waiting* if it cannot be ready or running because it is waiting at an exchange for a message. A *suspended* task is one that is not permitted to run or compete for system resources until it is resumed. The relationships between the task states are illustrated in Figure 3.

Priority

Each task has associated with it a priority that indicates its importance relative to other tasks in the system and relative to the interrupts of peripheral devices. RMX/80 schedules a task for execution based on the task's priority. Whenever a decision must be made on which task should be

run, the highest priority ready task is chosen. Each of the eight hardware interrupt levels has a set of priorities, one of which must be assigned to the task that services the interrupt. When an interrupt occurs that task is executed if it is the highest priority ready task. At the time a higher priority task preempts a lower priority task, RMX/80 saves all the relevant information about the pre-empted task so it can eventually resume execution as though it were never interrupted. This process is known as a context save.
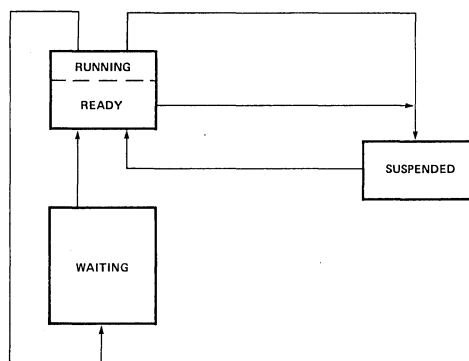
Figure 3. Task States

NUCLEUS OPERATIONS

The RMX/80 nucleus provides several operations that you can access with programmed calls. Two basic operations are covered in this section (additional operations are described in the RMX/80 User's Guide):

- RQSEND, send a message to an exchange
- RQWAIT, wait for a message or time interval

These two operations provide the capability to pass messages between tasks in a system running under RMX/80.

Message Format

The messages used by the send and wait operations to convey information between tasks are variable in length and contain the information shown in Figure 4.
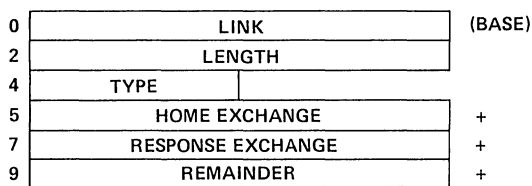
+Indicates optional

Figure 4. Message Format

Fields

1. LINK — a 2-byte field used to enter the message on a linked list at an exchange.

2. LENGTH — a 2-byte field containing the total length of the message in bytes. The minimum message length is 5 bytes (LINK, LENGTH, and TYPE).

3. TYPE — a 1-byte field indicating the type of message.

4. HOME EXCHANGE — an optional 2-byte field containing the address of an exchange to which this message should be sent when it has no further use. This field is very useful in implementing and managing a pool of messages.

5. RESPONSE EXCHANGE — an optional 2-byte field containing the address of an exchange to which a logical response to this message should be sent. This field is intended to specify the exchange at which a sending task is waiting for an acknowledgement message if one is needed.

6. REMAINDER — an optional field of arbitrary length that may contain any data portion of the message.

Sending a Message to an Exchange

The RQSEND operation enables a task to post a message at an exchange. When you send a message to an exchange, RMX/80 actually posts only the address of the message at the exchange, not the body of the message. RMX/80 avoids the overhead required to move an entire message to an exchange. Thus it is possible to queue a number of messages at the same exchange with little overhead in either execution time or memory requirements. When a task sends a message to an exchange, several functions are performed.

- The message is placed on the specified exchange.
- If there are one or more tasks waiting at the exchange, the first task is given the message and is made ready.
- If a higher priority task is thereby made ready, the sending task loses control until it once again becomes the highest priority ready task.

After a message is sent to an exchange, it must not be modified by the sending task. A task which then receives the message by waiting at the exchange where the message has been posted is free to modify the message. The format of the RQSEND operation is as follows.

RQSEND(exchange-address,message-address)

Message exchanges are defined by the user, and are normally addressed symbolically. For example, the exchange used to pass readings from an analog-to-digital (A/D) task might be named ATODEX. The reading itself could be contained in a message with the name RDNG. Thus, a typical call for a send in a PL/M program might be as follows:

CALL RQSEND (.ATODEX,.RDNG) ;

The call procedure in assembly language is as follows.

```
LXI    B,ATODEX

LXI    D,RDNG

CALL   RQSEND
```

The assembly language rules for passing parameters to RMX/80 are the same as for passing parameters to a PL/M procedure called from an assembly language module. For 2-byte parameters, the first parameter is passed in the B and C registers; the second parameter is passed in the D and E registers.

Waiting for a Message or Time Interval

The RQWAIT operation causes a task to wait for a message to arrive at an exchange. It is also possible to delay execution of a task when no message is anticipated for the task. The task simply waits for the desired time period at a message exchange where no message is ever sent. When a task waits for a message at an exchange several operations are performed.

- The task is made to wait until a message is sent to the specified exchange, or until the time limit has expired.
- When a message is available, its address is returned to the task.
- If the time limit expires before a message becomes available, a system TIME$OUT message is returned to the task.

The format of the RQWAIT operation is as follows.

RQWAIT(exchange-address,time-limit)

The time limit is entered as some number of system time units (50 milliseconds); a 1-second wait is equal to 20 time units. If zero is specified the wait is not timed, producing an indefinite wait until a message is actually sent to the exchange. Note that a specified wait of five time units may sometimes only produce an actual wait of four time units. This can occur if you enter a wait immediately before the clock "ticks." In this case the count would be decremented immediately after entering the wait. Only four full time unit periods would lapse before completion of the wait. Thus a user who wishes to ensure that at least five time units are spent in an asynchronous wait must specify six time units in the wait operation. A task which waits synchronously to the system clock, i.e., performs repetitive timed waits, does not have this problem because a new wait is executed following a tick that satisfied the previous wait. The following are typical calls for the RQWAIT operation.

PL/M

PTR = RQWAIT(.ATODEX,20) ;

The RQWAIT procedure returns an address value which is the address of a message.

Assembly Language

```
LXI    B,ATODEX

LXI    D,20

CALL   RQWAIT
```

The address of a message is returned in the HL register pair.

Send — Wait Interaction

To a large extent, the power of RMX/80 as a programming tool is derived from the interaction between send and wait. The interaction includes three multi-tasking operations.

- Communication
- Synchronization
- Mutual Exclusion

In describing these operations, a graphic notation for diagramming tasks, exchanges, and their interaction (send and wait operations) is useful. The notation is described in the next section on communication.

Communication. The most common interaction between tasks is that of communication — the transmission of data from one task to another via an exchange (Figure 5).
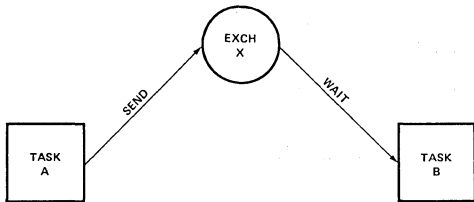
Figure 5. Communication

Rectangles designate tasks while circles represent exchanges. Arrows that are directed from tasks to exchanges indicate send operations. Wait operations are shown by arrows directed from exchanges to tasks.

Figure 5 shows an example of communication between task A and task B. Task A sends a message to exchange X and task B waits for a message at that exchange. Task A is the message producer and task B the message consumer.

Synchronization. At times there is a requirement to send a synchronizing signal from one task to another. This signal can take the form of a message that contains only header information, that is, LINK, LENGTH, and TYPE.

Let us consider the implementation of a task scheduler, used for the purposes of synchronizing another task that performs a particular function at periodic intervals. The relationship between the tasks and exchanges is shown in Figure 6.
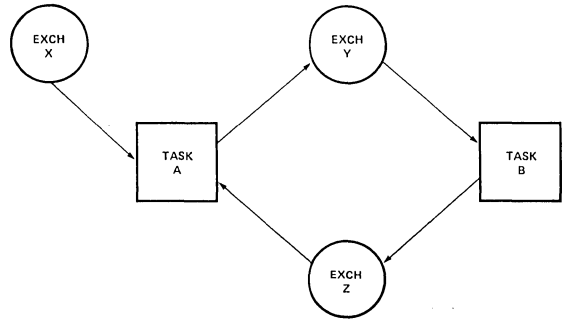
Figure 6. Synchronization

Task A, the scheduler, performs a timed wait on the X exchange. Note that the full wait period will always occur because there is no task that is sending messages to exchange X. In this manner, a specific timed wait by task A precedes the passing of a synchronization message from task A to task B via exchange Y, and then the return from task B to task A via the Z exchange.

If task B waited on X directly, rather than using task A for scheduling, it would be scheduled n system time units from when it waits instead of n units from the last time it was awakened. A comparison between the two methods is shown in Figure 7.
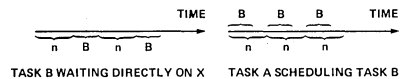
Figure 7. Scheduling Methods

Mutual Exclusion. In an environment with multi-tasking, resources often must be shared. Examples of shared resources include data structures and peripherals such as the Intel SBC 310 Math Module. Mutual exclusion can be used to ensure that only one task has access to a shared resource at a time. Figure 8 shows how an exchange can be used to limit access to a resource.
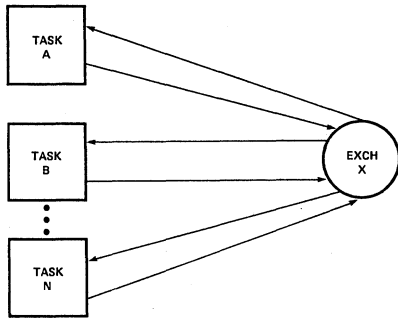
Figure 8. Mutual Exclusion

In this example, the X exchange is sent a single message at system initialization. Then, as tasks require the resource, they wait for a message from the X exchange. When the message is received, the task knows it has sole access to the resource because there is only one message associated with the exchange. After the task finishes with the resource it sends the message back to the X exchange. The next task waiting for the resource continues, knowing it has exclusive access to the resource.

EXTENSIONS

RMX/80 has several extensions which provide operations commonly used in real-time applications. The nucleus of RMX/80 requires less than two thousand bytes of memory and includes all of the basic operations. The extensions include a Free Space Manager, Terminal Handler, Disk File System, and a Debugger.

FREE SPACE MANAGER

The Free Space Manager maintains a pool of free RAM and allocates memory from that pool upon request from a task. The Free Space Manager also reclaims memory and returns it to the pool when it is no longer needed.

The Free Space Manager is especially useful in two applications. The first application arises from the need for variable length messages. If you have a task that produces messages of variable length, such as a task sending text for display on a CRT, the Free Space Manager can be used to provide a message to meet your exact size requirements.

An alternate solution is to maintain a pool of large fixed length messages. The pool can be maintained without the Free Space Manager; however, memory is wasted because of the unused space remaining in the fixed length messages.

The second application of the Free Space Manager relates specifically to effective use of memory. In a typical application, the total RAM requirement is computed by adding up the maximum RAM requirements for each task in a system as shown in Figure 9.

$RAM_{TOTAL} = MAX_A + MAX_B + MAX_C$
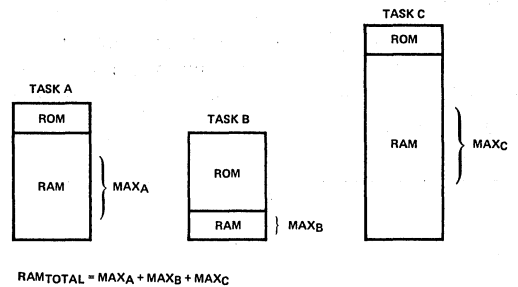
Figure 9. RAM Requirements

The efficiency of memory utilization is a function of the total RAM memory needed during typical system operation. Reducing the total amount of RAM by sharing it among the tasks often has little impact on total performance. However, significant cost advantages may be gained by reducing the total amount of memory. The memory requirements can be calculated as the minimum RAM for each task plus the pool (shared memory), as shown in Figure 10.

$RAM_{TOTAL} = MIN_A + MIN_B + MIN_C + POOL$
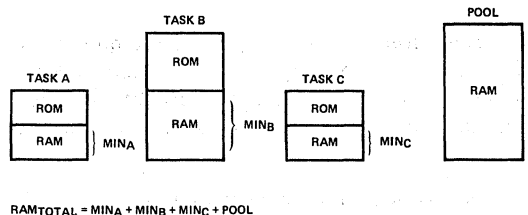
Figure 10. RAM Requirements Using a Pool

TERMINAL HANDLER

The Terminal Handler provides real-time asynchronous I/O between an operator terminal and tasks running under the RMX/80 executive. The Terminal Handler provides a line-edit capability similar to that of ISIS-II and an additional type-ahead feature. (ISIS-II is the diskette supervisor system used on the Intellec Microcomputer Development System.) Access to the Terminal Handler is provided by two exchanges where messages are sent to initiate read and write requests.

Several features of the Terminal Handler have been incorporated specifically to facilitate interaction with the debugger. Because of this interaction, the Terminal Handler is required for operation of the debugger.

DISK FILE SYSTEM

The Disk File System (DFS) provides users of RMX/80 with disk file management capabilities. This system allows user tasks to create, access, and maintain disk files in a real-time environment. This means that many I/O requests can be processed concurrently, rather than one at a time.

In addition to the file handling services, DFS provides a program loading facility that allows you to load program segments into memory from disk.

The DFS can be configured to include only those functions which you require. For example, if your disk accesses are sequential rather than random, you omit the SEEK function. This philosophy of minimizing memory requirements by including only the functions your application requires is found in virtually all aspects of RMX/80.

DEBUGGER

An environment that is continually changing in response to asynchronous physical events can present a serious debugging challenge. The Debugger aids you in debugging tasks running under the RMX/80 executive. The Debugger provides a command language that can be used to passively display information about the system, or actively modify and interact with the system.

Passive Functions

Because RMX/80 manages a fairly complex set of data structures, the Debugger has the capability of displaying them in an intelligible format. The Debugger can be used in this manner to view tasks, exchanges, messages, and other data structures maintained within the RMX/80 environment. The contents of all RAM and ROM memory locations may also be displayed by the Debugger.

Active Functions

The active Debugger functions include those of modifying memory, setting breakpoints, and monitoring stack overflow. The memory modification commands enable you to update the contents of memory and to move a series of bytes from any location to any other location.

Breakpoints can be set, allowing you to gain control when encountered by a task. Two kinds of breakpoints are supported: execution breakpoints and exchange breakpoints. An execution breakpoint can be placed at any instruction within read/write (RAM) memory. When the breakpoint is reached, the task encountering the breakpoint is stopped from further execution. The task registers may then be examined and modified before resuming execution.

Exchange breakpoints can be used to detect RQSEND and/or RQWAIT operations performed on specified exchanges. The exchange breakpoint can thus enable you to monitor the activity of any of the exchanges in your system. The task executing the appropriate RQSEND or RQWAIT to an exchange which has a breakpoint is stopped, allowing you to examine the task. This enables you to breakpoint a ROM resident task. The breakpointed task and the message involved in the operation with the exchange may then be displayed and modified before resuming execution.

The debugger can also be used to monitor stack overflow. This function is provided by the Debugger SCAN command which examines the stacks of all tasks in the system at a specified periodic interval. The fact that each task's stack is initialized with a unique value allows stack overflow to be detected. When a task stack overflows, it is removed from the system and a message is displayed.

USING RMX/80

This section of the application note describes the steps involved in using RMX/80. The process begins with the definition of the individual tasks and exchanges in your application. It continues with a discussion of the data structures that you must prepare. The task coding, compilation or assembly, linking, and locating is also described.

Finally, some comments are directed towards debugging tasks within the RMX/80 envirnment.

Before the details of using RMX/80 are discussed, some general observations are necessary to determine the suitability of RMX/80 for your application. To effectively utilize RMX/80, your application must either use interrupts or require device polling. Thus, the key element is the need to respond to external events. If your application satisfies this criteria, it is a likely candidate. However, you must then determine if RMX/80 is capable of supporting your application. This can be done by examining your interrupt response time and frequency requirements. The time required to transform an interrupt into a message that is sent to an interrupt exchange is approximately 800 microseconds for an SBC 80/20. This is the RMX/80 interrupt latency. It can be reduced to 60 microseconds by handling the interrupt directly, using the RQSETV operation to bypass the RMX/80 interrupt exchange mechanism. In this latter mode, an interrupt-driven asynchronous block transfer rate of about 10 kHz can be achieved.

TASK AND EXCHANGE DEFINITION

The initial design step for an application that runs under the RMX/80 Executive is to define your tasks, exchanges, and the interaction between them. This is perhaps best accomplished using the graphic notation introduced earlier in the section on Send – Wait Interaction. The graphic notation provides a clear picture of the relationships between the tasks and exchanges in your system. You can begin either in a top-down or bottom-up fashion. That is, you can use a top-down approach to define, at a gross level the operation of your system and then gradually break it down to the individual tasks. Or, you can start with the tasks associated with the external events in your application and then build the pieces to form the gross structure of your system.

The bottom-up approach forces you to begin with external events that drive your system. The number of these events, the amount of processing required, and the relationships between them define the tasks and exchanges in a system. For example, consider a system that samples an analog input with an A/D converter. Assume that the A/D provides an interrupt at the completion of a conversion. To use the data from the A/D converter it may also be necessary to scale it and add an offset.

With this information the portion of the task and exchange definition that relates to this function can be constructed.

Begin with the external event, the interrupt from the A/D. An ·interrupt priority level must be assigned to the A/D converter. This same level will be used by the task which waits on the interrupt exchange.

The relationship between the interrupt exchange and the A/D task is shown in Figure 11. If processing must be performed on raw data from the A/D, a second, lower priority, task could be used. Another task for this function will require a synchronizing signal from the ADC task to indicate that raw A/D data has been obtained and is ready for processing.
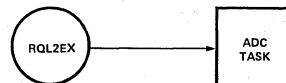
Figure 11. Interrupt Exchange and A/D Task

The interaction between the ADC task and the CONV task that processes the raw A/D data is shown in Figure 12. Two exchanges provide synchronization. The ADC task uses the TRGR exchange to signal that data is ready for processing by the CONV task. The CONV task uses the RTRGR exchange to signal the completion of its processing and thus its readiness to accept more raw data.
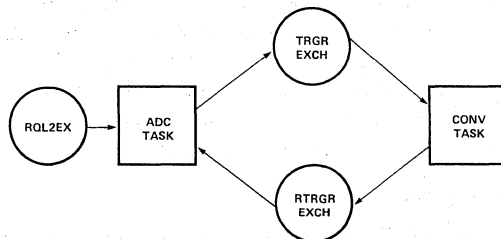
Figure 12. ADC and CONV Task Interaction

In this example two tasks and three exchanges have been defined. To develop an entire system, the tasks and exchanges associated with all of the external events in the system can be defined in the same manner. Then, proceeding bottom-up, the next step is to define the tasks and exchanges required to support the interaction between tasks running at the level of the real-time events.

After defining the entire application, you can begin actual coding of the tasks. You may choose to code in either assembly language or the PL/M 80 high-level language. Where possible, it is desirable to code in PL/M because PL/M lends itself to structured programming. Assembly language often encourages an ad hoc approach. Even if your application ultimately requires assembly language coding because of critical time and/or space parameters, initial design work in PL/M followed by translation into assembly language is recommended.

A total of 15 operations are supported by the RMX/80 nucleus. Only two of the operations, RQSEND and RQWAIT, are described in any detail in this application note. The remaining operations are described in the *RMX/80 User's Guide*. The reason for presenting only the send and wait operations is because they are sufficient for the implementation of a large number of real-time applications. These two operations provide a great deal of power and flexibility, yet their simplicity should enable those who are new to real-time programming to quickly develop applications.

PRIORITY ASSIGNMENT

The relative priority of tasks within a system running under RMX/80 determines which task is to be executed. Therefore, the assignment of a priority to each task is extremely crucial. For example, consider a compute bound task placed at a higher priority than an interrupt-driven task responsible for servicing an I/O device. This improper assignment of priorities could result in missed interrupts from the I/O device. Several steps can be followed in the assignment of task priorities.

1. Assign hardware interrupt priority levels according to the requirements of your application.

2. Specify priorities for the tasks which service the interrupts. These tasks should generally be short and serve only to perform the data

transfers. A second task with a priority lower than those assigned to the hardware interrupts should be used where further processing of the data is required.

3. Priority assignment should be made for all other tasks in the system based on the relative importance and interaction among the tasks.

Unfortunately the last step in assigning task priorities is largely intuitive. In fact, you may need some empirical data from actually running your application before you settle on your final task priority assignment.

STATIC DESCRIPTORS

When a system running under RMX/80 begins execution, several tables of data are used to initialize the system. These tables usually reside in ROM. The first table is the create table (RQCRTB) that specifies the number of tasks and exchanges in the system, and the addresses of the initial task table and the initial exchange table. The initial exchange table contains the addresses of all the exchange descriptors. The initial task table contains the static task descriptors for each task, and contains the following task parameters.

1. Name

2. Initial PC — the location at which to start task execution

3. Initial SP — the location at which to start the task stack

4. Stack length

5. Priority

6. Initial Exchange (described in the *RMX/80 User's Guide*)

7. TD Address — the RAM address of the task descriptor

You must prepare all three of these tables to produce a configuration module for RMX/80. The release diskette for RMX/80 includes a set of files which contain assembly language macros that simplify the preparation of your configuration module. The relationship between these tables is shown in Figure 13.

COMPILATION/ASSEMBLY

Preparing program segments for compilation and assembly can be simplified by use of files provided on the RMX/80 diskette. As described in the last

section, a set of macros is included to assist you in preparing your configuration module. Other files are provided that are useful when coding calls to RMX/80 and preparing data structures in PL/M.
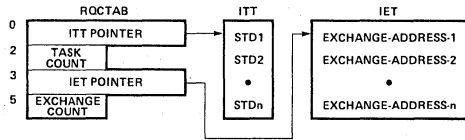
Figure 13. ROM Based Tables

By coding in a modular fashion you can separately compile and maintain tasks. This is advisable since a single large module containing all your tasks would require a lengthy recompilation to change any one of the tasks. Following the compilation and assembly of your source code modules, a library containing the object modules can be created.

LINKING

The process of linking prepares a single object module from libraries containing the RMX/80 object modules and your own application libraries or separate object modules. The order in which you specify the files to be linked is crucial to successful linking. In general, your libraries or separate object modules should be specified before the RMX/80 libraries. The link command should conclude with the unresolved library (UNRSLV.LIB) that contains miscellaneous modules for resolving PUBLICs not used in the application code. PUBLICs extend the scope of variables to allow linkage between separate program modules. Figure 14 illustrates how an application program is linked from RMX/80 and user tasks.

LOCATING

It is appropriate in this section to give some guidelines regarding the assignment of RAM and ROM address space for your Single Board Computer environment. The SBC 80 Single Board Computers have ROM based at location 0. Since the LOCATE program places all code in a contiguous block, the code must begin at location 0. Likewise, the read/write (RAM) data is also placed in a contiguous block. The base address of data should be placed at your RAM base address. Depending on the

amount of code space required by your application it may be necessary to move the RAM memory base address on your SBC to a higher location. A STACKSIZE of zero should be specified because you allocate stack for each RMX/80 task in the static task descriptors.

DEBUGGING

As mentioned in the overview of the RMX/80 Debugger, the real-time environment is a complex one in which to debug your programs. Intel provides two tools that you can use for debugging. The RMX/80 Debugger and the Intel In-Circuit Emulator (ICE). It is desirable to have both of these debugging tools at your disposal.
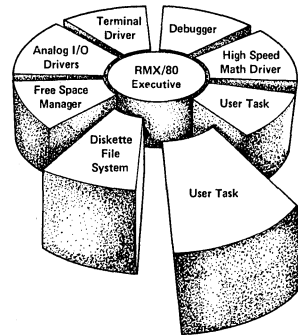
Figure 14. RMX/80 Linking

ICE enables you to use Intel Microcomputer Development System memory in place of SBC 80 memory. This allows RAM residency during your debugging as opposed to programming PROMs for each iteration. Your system may initially fail before the RMX/80 Debugger can begin operation. In this situation ICE can be used to debug your program.

APPLICATIONS

RMX/80 is suitable for a wide variety of applications. Two specific examples are presented in this application note. Each example illustrates the steps involved in using RMX/80 and provides a detailed description of the coding itself.

MINIMAL TERMINAL HANDLER

The basic functions required for a terminal handler are well defined. The handler must respond to

operator input, transmit output characters, and echo characters as they are entered. This application note describes one implementation of a minimal terminal handler.

The terminal handler presented here is not the RMX/80 Terminal Handler. It does provide some common functions and uses the same exchanges and message formats. However, many features of the RMX/80 Terminal Handler have been left out. Omitted features include special hooks to run with the Debugger, an alarm exchange, control S, Q, and O operations.

As described in the chapter on using RMX/80, the process of developing an RMX/80 application begins with the definition of the tasks and exchanges. The graphic notation is used to prepare a diagram (Figure 15) showing the tasks, exchanges, and their interaction.
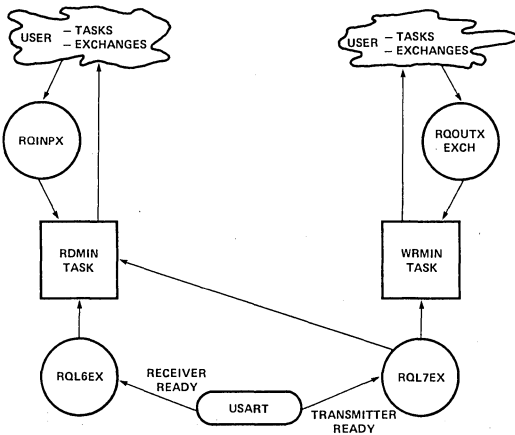
Figure 15. Minimal Terminal Handler

As shown in Figure 15, the RDMIN task waits on the RQINPX exchange for input requests. The RDMIN task also successively waits on the RQL6EX and RQL7EX exchanges. It uses the RQL6EX exchange to determine when a character has been received by the USART. The RQL7EX exchange is used to indicate when the transmitter is ready to accept another character. RDMIN uses RQL7EX for echoing input characters.

The WRMIN task waits on the RQOUTX exchange for output requests. When it receives a request, it

waits on the RQL7EX to determine when characters can be sent to the USART.

The following listing* shows the RDMIN and WRMIN tasks. These tasks provide a minimal terminal handler. The program is written in PL/M. The WRMIN task is also presented in assembly language in Appendix B. The program listing is interspersed with explanatory text. The program begins with the program segment label "MINIMAL$TERMINAL$HANDLER:" and a DO statement.

```
1       MINIMAL$TERMINAL$HANDLER:
        DO;
```

Several macros are declared using the reserved word LITERALLY. These macros are expanded at compile time by textual substitution.

```
2   1   DECLARE TRUE LITERALLY '0FFH';
3   1   DECLARE FOREVER LITERALLY 'WHILE TRUE';

        /* SPECIAL ASCII CHARACTERS */
4   1   DECLARE
        BELL          LITERALLY '07H',
        LF            LITERALLY '0AH',
        CR            LITERALLY '0DH',
        CONTROL$R     LITERALLY '12H',
        CONTROL$X     LITERALLY '18H',
        ESC           LITERALLY '1BH',
        RUBOUT        LITERALLY '7FH';
```

Some macros are used to simplify the declaration of RMX/80 data structures. The structures declared here are for the exchange descriptor, interrupt exchange descriptor, and the messages used by the minimal terminal handler.

```
5   1   DECLARE EXCHANGE$DESCRIPTOR LITERALLY 'STRUCTURE (
        MESSAGE$HEAD ADDRESS,
        MESSAGE$TAIL ADDRESS,
        TASK$HEAD ADDRESS,
        TASK$TAIL ADDRESS,
        EXCHANGE$LINK ADDRESS)';

6   1   DECLARE INT$EXCHANGE$DESCRIPTOR LITERALLY 'STRUCTURE (
        MESSAGE$HEAD ADDRESS,
        MESSAGE$TAIL ADDRESS,
        TASK$HEAD ADDRESS,
        TASK$TAIL ADDRESS,
        EXCHANGE$LINK ADDRESS,
        LINK ADDRESS,
        LENGTH ADDRESS,
        TYPE BYTE)';

7   1   DECLARE TH$MSG LITERALLY 'STRUCTURE(
        LINK ADDRESS,
        LENGTH ADDRESS,
        TYPE BYTE,
        HOME$EXCHANGE ADDRESS,
        RESPONSE$EXCHANGE ADDRESS,
        STATUS ADDRESS,
        BUFFER$ADDRESS ADDRESS,
        COUNT ADDRESS,
        ACTUAL ADDRESS)';
```

The following macros are specifically for the SBC 80/20. The macros require changes to run the minimal terminal handler on a different Single Board Computer. Intel 8253 timer/counter and 8251 USART chips are used.

*Full size listings in Appendixes A and C.

```
        /*
          8253 PORT ADDRESSES.
        */
 8   1  DECLARE A8253$MODE LITERALLY '0DFH';
 9   1  DECLARE A8253$CTR2 LITERALLY '0DEH';

        /*
          8253 COMMANDS.
        */
10   1  DECLARE SELECT$2 LITERALLY '10000000B';
11   1  DECLARE RL$BOTH  LITERALLY '00110000B';
12   1  DECLARE MODE$3 LITERALLY '00000110B';
13   1  DECLARE B2400 LITERALLY '001CH';

        /*
          8251 PORT ADDRESSES.
        */
14   1  DECLARE USART$IN LITERALLY '0ECH',
                USART$OUT LITERALLY '0ECH',
                USART$CONTROL LITERALLY '0EDH';

        /*
          8251 MODES.
        */
15   1  DECLARE STOP$1  LITERALLY '01000000B';
16   1  DECLARE CL8 LITERALLY '00001100B';
17   1  DECLARE RATE$16X LITERALLY '00000010B';

        /*
          8251 COMMANDS.
        */
18   1  DECLARE USART$RESET LITERALLY '01000000B',
                RTS        LITERALLY '00100000B',
                ERROR$RESET LITERALLY '00010000B',
                RXE        LITERALLY '00000100B',
                DTR        LITERALLY '00000010B',
                TXEN       LITERALLY '00000001B';
```

RDMIN and WRMIN call three RMX/80 operations. They are RQSEND, RQWAIT, and RQELVL. RQSEND and RQWAIT allow tasks to send and receive messages from exchanges. RQELVL enables a specific interrupt level.

```
19   1  RQSEND:
20   2    PROCEDURE (EXCHANGE$POINTER,MESSAGE$POINTER) EXTERNAL;
21   2    DECLARE (EXCHANGE$POINTER,MESSAGE$POINTER) ADDRESS;
         END RQSEND;

22   1  RQWAIT:
23   2    PROCEDURE (EXCHANGE$POINTER,DELAY) ADDRESS EXTERNAL;
24   2    DECLARE (EXCHANGE$POINTER,DELAY) ADDRESS;
         END RQWAIT;

25   1  RQELVL:
26   2    PROCEDURE (LEVEL) EXTERNAL;
27   2    DECLARE LEVEL BYTE;
         END RQELVL;
```

The exchange descriptors and interrupt exchange descriptors must be PUBLIC because they are referenced by the configuration module.

```
28   1  DECLARE RQINPX EXCHANGE$DESCRIPTOR PUBLIC;
29   1  DECLARE RQOUTX EXCHANGE$DESCRIPTOR PUBLIC;

30   1  DECLARE RQL6EX INT$EXCHANGE$DESCRIPTOR PUBLIC;
31   1  DECLARE RQL7EX INT$EXCHANGE$DESCRIPTOR PUBLIC;
```

The following procedure initializes the 8253 and 8251 (USART). The 8253 generates the baud rate clock (2400 baud in this example). The program sends four nulls to the USART control port to ensure that the USART is ready for a command, no matter what state it was previously in. The program then sends a reset command to the USART, followed by the mode and another command.

```
32   1  INITIALIZATION:
         PROCEDURE;
33   2    OUTPUT(A8253$MODE) = SELECT$2 OR RL$BOTH OR MODE$3;
34   2    OUTPUT(A8253$CTR2) = LOW(B2400);

35   2    OUTPUT(A8253$CTR2) = HIGH(B2400);
36   2    OUTPUT(USART$CONTROL),
           OUTPUT(USART$CONTROL),
           OUTPUT(USART$CONTROL),
           OUTPUT(USART$CONTROL) = 0;
37   2    OUTPUT(USART$CONTROL) = USART$RESET;
38   2    OUTPUT(USART$CONTROL) = STOP$1 OR CL8 OR RATE$16X;
39   2    OUTPUT(USART$CONTROL) = RTS OR ERROR$RESET OR
                                  RXE OR DTR OR TXEN;
40   2  END INITIALIZATION;
```

Tasks coded in PL/M take the form of parameterless PUBLIC procedures. The procedure declaration is followed by the variables used in RDMIN. MSGPTR receives the address of an input request message. The based-variable MSG accesses the data in the input request message. INTMSG is a dummy variable which simply receives the address of the interrupt message. BUF$ADDRESS points to the buffer where the input characters are to be placed. The BUF array is based at the buffer pointed to by BUF$ADDRESS.

```
41   1  RD$MIN:
         PROCEDURE PUBLIC;
42   2    DECLARE (MSGPTR,INTMSG,BUF$ADDRESS) ADDRESS;
43   2    DECLARE (CHAR,PTR,I) BYTE;
44   2    DECLARE MSG BASED MSGPTR TH$MSG;
45   2    DECLARE (BUF BASED BUF$ADDRESS) (1) BYTE;
```

The RDMIN task echoes characters after they are read in. The ECHO$CHAR procedure performs this function. It waits for a level 7 interrupt, indicating that the transmitter is ready for another character. ECHO$CHAR then transmits the character.

```
46   2  ECHO$CHAR:
         PROCEDURE (CHAR);
47   3    DECLARE CHAR BYTE;
48   3    INTMSG = RQWAIT(.RQL7EX,0);
49   3    OUTPUT(USART$OUT) = CHAR;
50   3    END ECHO$CHAR;
```

Execution of the RDMIN task starts with the next statement, a call to the initialization procedure. This call is followed by two calls to the procedure which will enable interrupt levels 6 and 7.

```
51   2    CALL INITIALIZATION;

52   2    CALL RQELVL(6);
53   2    CALL RQELVL(7);
```

The basic structure of an RMX/80 task is that of a program with an imbedded infinite loop. This loop starts with the DO FOREVER statement. In the continuous loop, the task waits for an input request message. This wait is satisfied when some other task in the system sends an input request message to the RQINPX exchange. The based variable used to point to BUF is assigned from a field in the input request message, MSG.BUFFER$ADDRESS. An index for the BUF array, PTR, and the variable CHAR are initialized.

```
54   2    DO FOREVER;
55   3      MSGPTR = RQWAIT(.RQINPX,0);
56   3      BUF$ADDRESS = MSG.BUFFER$ADDRESS - 1;
57   3      PTR = 0;
58   3      CHAR = NOT CR;
```

Task execution continues inside the next loop until a carriage return (CR) is input. An escape character (ESC) within the loop simulates a CR

which enables an exit from the loop. The task simply waits on the RQL6EX exchange for a message. This amounts to an interrupt service routine. When the wait is satisfied, the USART has received a character.

```
59   3        DO WHILE CHAR <> CR;
60   4          INTMSG = RQWAIT(.RQL6EX,0);
```

The next statement performs a whole series of operations. The character input from the USART is logically ANDed with 7FH to mask off the parity bit, assigned to the variable CHAR, and tested to determine if it is a RUBOUT character. If a RUBOUT is found, either a BELL is echoed to the terminal if there are not characters to delete in the buffer (PTR = 0), or the last character in the buffer is echoed and the pointer is decremented.

```
61   4        IF (CHAR := INPUT(USART$IN) AND 7FH) = RUBOUT THEN
62   4          DO;
63   5            IF PTR = 0 THEN
64   5              CALL ECHO$CHAR(BELL);
65   5            ELSE
66   6            DO;
67   6              CALL ECHO$CHAR(BUF(PTR));
                    PTR = PTR - 1;
68   6            END;
69   5          END;
```

If CHAR is not a RUBOUT, it is tested for a CONTROL$X. The function of a CONTROL$X is to delete the entire line by resetting PTR to zero. After deleting the line, the system prompts the operator with a "#" character and is ready to accept a new line.

```
              ELSE
70   4        DO;
71   5          IF CHAR = CONTROL$X THEN
72   5          DO;
73   6            CALL ECHO$CHAR('#');
74   6            CALL ECHO$CHAR(CR);
75   6            CALL ECHO$CHAR(LF);
76   6            PTR = 0;
77   6          END;
```

The next test determines if CHAR is a CONTROL$R. CONTROL$R echoes the entire line that has been entered. This function is useful for displaying a line containing a number of RUBOUTs. Such lines can be difficult to interpret because RUBOUT echoes deleted characters. Because CONTROL$R echoes only the remaining data in the buffer, it eliminates "garbage" from the display.

```
              ELSE
78   5        DO;
79   6          IF CHAR = CONTROL$R THEN
80   6          DO;
81   7            CALL ECHO$CHAR(CR);
82   7            CALL ECHO$CHAR(LF);
83   7            DO I = 1 TO PTR;
84   8              CALL ECHO$CHAR(BUF(I));
85   8            END;
86   7          END;
```

The character is then placed in the buffer unless the end of the buffer has been reached. If the buffer is full, a BELL is sent to the terminal.

```
              ELSE
87   6        DO;
88   7          IF PTR < MSG.COUNT THEN
89   7            BUF(PTR := PTR+1) = CHAR;
              ELSE
90   7          DO;
91   8            IF CHAR <> CR THEN
92   8              CHAR = BELL;
93   8          END;
```

The last test is for an ESC character. It is echoed as a "$" and is treated as if a CR were entered.

```
94   7          IF CHAR = ESC THEN
95   7          DO;
96   8            CALL ECHO$CHAR('$');
97   8            CHAR = CR;
98   8          END;
99   8          CALL ECHO$CHAR(CHAR);
100  7        END;
101  6      END;
102  5      END;
103  4    END;
```

The program places a line feed (LF) at the end of the buffer when an exit is forced by a CR or an ESC. The input request message actual character count (MSG.ACTUAL) and the status (MSG. STATUS) are set before sending the message to its response exchange.

```
104  3    IF PTR < MSG.COUNT THEN
105  3      BUF(PTR:=PTR+1) = LF;
106  3    MSG.ACTUAL = PTR;
107  3    MSG.STATUS = 0;
108  3    CALL RQSEND(MSG.RESPONSE$EXCHANGE,MSGPTR);
109  3    CALL ECHO$CHAR(LF);
110  3  END;
111  2  END RD$MIN;
```

The WRMIN task begins by enabling interrupt level 7. Note that no other initialization is performed before WRMIN waits for an output request message to arrive at the RQOUTX exchange. Here correct operation depends on the fact that RDMIN has a higher priority than WRMIN. Were this not the case, WRMIN could try to transmit a message before the 8253 and 8251 have been set up.

```
112  1  WR$MIN:
        PROCEDURE PUBLIC;
113  2    DECLARE (MSGPTR,INTMSG,BUF$ADDRESS) ADDRESS;
114  2    DECLARE PTR BYTE;
115  2    DECLARE MSG BASED MSGPTR TH$MSG;
116  2    DECLARE (BUF BASED BUF$ADDRESS) (1) BYTE;
117  2    CALL RQELVL(7);
118  2    DO FOREVER;
119  3      MSGPTR = RQWAIT(.RQOUTX,0);
120  3      BUF$ADDRESS = MSG.BUFFER$ADDRESS - 1;
```

The next loop transmits all of the characters specified by the output request message. Once again, the interrupt service routine is implemented by simply waiting on the RQL7EX exchange for a transmitter ready interrupt message. When this message is received, the next character in the buffer is transmitted.

```
121  3              DO PTR = 1 TO MSG.COUNT;
122  4                 INTMSG = RQWAIT(.RQL7EX,0);
123  4                 OUTPUT(USART$OUT) = BUF(PTR);
124  4              END;
```

The WRMIN task concludes by setting the actual count and status, and then sends the output request message to its response exchange.

```
125  3              MSG.ACTUAL = MSG.COUNT;
126  3              MSG.STATUS = 0;
127  3              CALL RQSEND(MSG.RESPONSE$EXCHANGE,MSGPTR);
128  3           END;
129  2        END WRMIN;
```

Using the macros provided on the RMX/80 diskette, the following static task descriptors (STD) should be placed in your configuration module.

> STD   RDMIN,64,112,0
>
> STD   WRMIN,64,128,0

The entries in the STD are interpreted as follows.

> STD   NAME,STKLEN,PRI,EXCH

where:

NAME   = the symbolic name assigned to the task associated with the STD

STKLEN = the number of bytes allocated to the task stack

PRI   = the task priority level

EXCH   = an optional field, usually 0

Priorities of 112 and 128 have been assigned to RDMIN and WRMIN because they correspond to hardware interrupt levels 6 and 7.

The following exchange addresses should be placed in your configuration module.

> XCHADR   RQINPX
>
> XCHADR   RQOUTX
>
> XCHADR   RQL6EX
>
> XCHADR   RQL7EX

The XCHADR macro only requires the address of the exchange descriptor.

Characters typed at the terminal are ignored unless an input request message has been received. Thus, type-ahead is not a built-in feature. However, if type-ahead is desired, it is sufficient to ensure that input requests are always queued for the RDMIN task and that the full input buffers are sent to an exchange that queues full buffers.

This can easily be accomplished by sending several input requests to the RQINPX. These input requests have the address of a "full-buffer" exchange as the response exchange and the RQINPX exchange as the home exchange. Then, tasks needing terminal input wait on the "full-buffer" exchange and send the message to the home exchange when finished.

CLOSED-LOOP ANALOG CONTROL

In the next example, a closed-loop analog control subsystem using the Intel SBC 711 analog-to-digital board illustrates task scheduling and synchronization in a process control application. In general, the subsystem samples an analog input at specified intervals, converts the data to temperature in degrees centigrade, and then — based upon programmed temperature limits — controls a heating element. The algorithm used provides a 2-position controller with neutral intermediate zone (or simply "bang-bang" control). The control algorithm is shown in Figure 16.
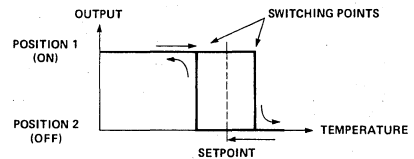
Figure 16. 2-Position Controller with Neutral Intermediate Zone

The graphic notation in Figure 17 diagrams the tasks, exchanges, and their interaction.
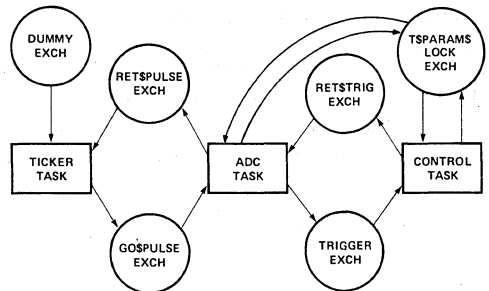
Figure 17. Analog Subsystem

This application includes three tasks and six associated exchanges. The TICKER task schedules the ADC task. TICKER has a very high priority because nothing else in the system should interfere with its scheduling activities. It is also a very short task since it repetitively executes a timed wait and then handshakes a message.

TICKER schedules the ADC task. The ADC task services the A/D converter. After obtaining data from the A/D it handshakes with the CONTROL task to signal that data is ready for processing. The ADC task is assigned a priority equivalent to the level of the hardware interrupt from the A/D. Clearly, calculations should not be performed at that priority.

Thus, CONTROL performs the processing function at a lower priority. The CONTROL task used the T$PARAM$LOCK exchange to govern access to the control parameters. This avoids problems resulting when some other task is updating the parameters at the same time the CONTROL task is using them for testing.

As in the minimal terminal handler, the following listing contains the complete analog subsystem tasks and is interspersed with explanatory text. The program begins with the program segment label "ATOD:" and a DO statement.

```
1            ATOD:
             DO;
```

The macros and externals used in this module are brought in by means of INCLUDEs from the RMX/80 diskette.

```
             $INCLUDE(:F1:COMMON.ELT)
2    1  =    DECLARE TRUE LITERALLY 'OFFH';
3    1  =    DECLARE FALSE LITERALLY 'OOH';
4    1  =    DECLARE BOOLEAN LITERALLY 'BYTE';
5    1  =    DECLARE FOREVER LITERALLY 'WHILE 1';

             $INCLUDE(:F1:EXCH.ELT)
6    1  =    DECLARE EXCHANGE$DESCRIPTOR LITERALLY 'STRUCTURE (
        =      MESSAGE$HEAD ADDRESS,
        =      MESSAGE$TAIL ADDRESS,
        =      TASK$HEAD ADDRESS,
        =      TASK$TAIL ADDRESS,
        =      EXCHANGE$LINK ADDRESS)';

             $INCLUDE(:F1:IED.ELT)
7    1  =    DECLARE INT$EXCHANGE$DESCRIPTOR LITERALLY 'STRUCTURE (
        =      MESSAGE$HEAD ADDRESS,
        =      MESSAGE$TAIL ADDRESS,
        =      TASK$HEAD ADDRESS,
        =      TASK$TAIL ADDRESS,
        =      EXCHANGE$LINK ADDRESS,
        =      LINK ADDRESS,
        =      LENGTH ADDRESS,
        =      TYPE BYTE)';

             $INCLUDE(:F1:MSG.ELT)
8    1  =    DECLARE MSG$HDR LITERALLY '
        =      LINK ADDRESS,
        =      LENGTH ADDRESS,
        =      TYPE BYTE,
        =      HOME$EXCHANGE ADDRESS,
        =      RESPONSE$EXCHANGE ADDRESS';

9    1  =    DECLARE MSG$DESCRIPTOR LITERALLY 'STRUCTURE(
        =      MSG$HDR,
        =      REMAINDER(1) BYTE)';
             $INCLUDE(:F1:INTRPT.EXT)
10   1  =    RQENDI:
        =      PROCEDURE EXTERNAL;
```

```
11   2  =    END RQENDI;
        =
12   1  =    RQELVL:
        =      PROCEDURE (LEVEL) EXTERNAL;
13   2  =      DECLARE LEVEL BYTE;
        =
14   2  =    END RQELVL;
        =
15   1  =    RQDLVL:
        =      PROCEDURE (LEVEL) EXTERNAL;
16   2  =      DECLARE LEVEL BYTE;
        =
17   2  =    END RQDLVL;
        =
18   1  =    RQSETV:
        =      PROCEDURE (PROC,LEVEL) EXTERNAL;
19   2  =      DECLARE PROC ADDRESS;
20   2  =      DECLARE LEVEL BYTE;
        =
21   2  =    END RQSETV;

             $INCLUDE(:F1:SYNCH.EXT)
22   1  =    RQSEND:
        =      PROCEDURE (EXCHANGE$POINTER,MESSAGE$POINTER) EXTERNAL;
23   2  =      DECLARE (EXCHANGE$POINTER,MESSAGE$POINTER) ADDRESS;
        =
24   2  =    END RQSEND;
        =
25   1  =    RQWAIT:
        =      PROCEDURE (EXCHANGE$POINTER,DELAY) ADDRESS EXTERNAL;
26   2  =      DECLARE (EXCHANGE$POINTER,DELAY) ADDRESS;
        =
27   2  =    END RQWAIT;
        =
28   1  =    RQACPT:
        =      PROCEDURE (EXCHANGE$POINTER) ADDRESS EXTERNAL;
29   2  =      DECLARE EXCHANGE$POINTER ADDRESS;
        =
30   2  =    END RQACPT;
        =
31   1  =    RQISND:
        =      PROCEDURE (IED$PTR) EXTERNAL;
32   2  =      DECLARE IED$PTR ADDRESS;
        =
33   2  =    END RQISND;
```

Additional macros are declared to aid in the use of the SBC 711 analog-to-digital board.

```
             /*
                SBC 711 ANALOG TO DIGITAL BOARD
             */
34   1       DECLARE ADC$BASE ADDRESS AT (0F700H);
35   1       DECLARE COMMAND$REGISTER BYTE AT (.ADC$BASE+0);
36   1       DECLARE STATUS$REGISTER BYTE AT (.ADC$BASE+0);
37   1       DECLARE FIRST$CHANNEL$REGISTER BYTE AT (.ADC$BASE+1);
38   1       DECLARE LAST$CHANNEL$REGISTER BYTE AT (.ADC$BASE+2);
39   1       DECLARE CLEAR$INTERRUPT$REQUEST BYTE AT (.ADC$BASE+3);
40   1       DECLARE ADC$DATA$REGISS BYTE AT (.ADC$BASE+4);

41   1       DECLARE GO$BIT LITERALLY '1';
42   1       DECLARE AUTO$INCREMENT$ENABLE LITERALLY '2';
43   1       DECLARE BUSY LITERALLY '8';
44   1       DECLARE EOS$INTERRUPT$ENABLE LITERALLY '10H';
45   1       DECLARE EOC$INTERRUPT$ENABLE LITERALLY '20H';
46   1       DECLARE END$OF$SCAN LITERALLY '40H';
47   1       DECLARE END$OF$CONVERSION LITERALLY '80H';
```

The exchange descriptors and the interrupt exchange descriptors are declared.

```
48   1       DECLARE DUMMY EXCHANGE$DESCRIPTOR PUBLIC;
49   1       DECLARE RET$PULSE EXCHANGE$DESCRIPTOR PUBLIC;
50   1       DECLARE GO$PULSE EXCHANGE$DESCRIPTOR PUBLIC;
51   1       DECLARE TRIGGER EXCHANGE$DESCRIPTOR PUBLIC;
52   1       DECLARE RET$TRIG EXCHANGE$DESCRIPTOR PUBLIC;
53   1       DECLARE RQL2EX INT$EXCHANGE$DESCRIPTOR;
```

The CONTROL task uses an external data structure to obtain operating parameters. This data structure (BOX$PARAMS) has an exchange associated with it (T$PARAM$LOCK) that is used to provide mutual exclusion, ensuring that only one task accesses the data structure at a time.

```
54   1       DECLARE T$PARAM$LOCK EXCHANGE$DESCRIPTOR EXTERNAL;

55   1       DECLARE BOX$PARAMS(5) STRUCTURE(
                 CHANNEL BYTE,
                 SET$POINT ADDRESS,
                 ERROR ADDRESS,
                 OFFSET ADDRESS,
                 SAMPLES ADDRESS,
                 COUNT ADDRESS,
                 ACCUM ADDRESS,
                 READING ADDRESS ) EXTERNAL;
```

TICKER, the scheduler task, has an initialization sequence in which it sets up two messages and sends them to the RET$PULSE exchange. Then it enters an infinite loop where it waits on the DUMMY exchange for 250 milliseconds. After the timed wait is complete, TICKER passes a message from the RET$PULSE exchange to the GO$-PULSE exchange. In effect this is a handshake, checking to see that the ADC task has completed its last operation and then signaling it to perform another.

```
56   1      TICKER$TASK:
              PROCEDURE PUBLIC;
57   2          DECLARE MSG ADDRESS;
58   2          DECLARE PULSE$MSG(2) STRUCTURE (
                    MSG$HDR );

59   2          PULSE$MSG(0).LENGTH,
                PULSE$MSG(1).LENGTH = SIZE(PULSE$MSG(0));
60   2          PULSE$MSG(0).TYPE,
                PULSE$MSG(1).TYPE = 65;
61   2          CALL RQSEND(.RET$PULSE,.PULSE$MSG(0));
62   2          CALL RQSEND(.RET$PULSE,.PULSE$MSG(1));

63   2          DO FOREVER;
64   3              MSG = RQWAIT(.DUMMY,5);
65   3              MSG = RQWAIT(.RET$PULSE,0);
66   3              CALL RQSEND(.GO$PULSE,MSG);
67   3          END;

68   2      END TICKER$TASK;
```

Scheduled by TICKER, the ADC task performs the A/D sampling. It begins by setting up TRIGGER$-MSG and enabling the level 2 interrupt from the A/D. Inside the ADC task continuous loop, messages are passed from the GO$PULSE exchange to the RET$PULSE exchange. Then it waits for access to the BOX$PARAMS data structure. When the ADC task has access, it loops through the A/D channels, accumulating readings in BOX$PARAMS. After all the A/D channels are sampled and the BOX$PARAMS readings updated, the LOCK$MSG is returned to the T$PARAM$LOCK exchange. The ADC task concludes the continuous loop by handshaking a message with the CONTROL task.

```
69   1      ADC$TASK:
              PROCEDURE PUBLIC;
70   2          DECLARE TRIGGER$MSG STRUCTURE (
                    MSG$HDR );
71   2          DECLARE (T$MSG,MSG,LOCK$MSG) ADDRESS;
72   2          DECLARE I BYTE;
73   2          DECLARE GAIN LITERALLY '00';
74   2          DECLARE N$CHNLS LITERALLY '5';

75   2          TRIGGER$MSG.LENGTH = SIZE(TRIGGER$MSG);
76   2          TRIGGER$MSG.TYPE = 65;
77   2          CALL RQSEND(.RET$TRIG,.TRIGGER$MSG);
78   2          CALL RQELVL(2);

79   2          DO FOREVER;
80   3              MSG = RQWAIT(.GO$PULSE,0);
81   3              CALL RQSEND(.RET$PULSE,MSG);
82   3              LOCK$MSG = RQWAIT(.T$PARAM$LOCK,0);
83   3              DO I = 0 TO N$CHNLS-1;
84   4                  FIRST$CHANNEL$REGISTER = BOX$PARAMS(I).CHANNEL
                            + ROL(GAIN,6);
85   4                  COMMAND$REGISTER = GO$BIT
                            OR EOC$INTERRUPT$ENABLE;
86   4                  MSG = RQWAIT(.RQL2EX,0);
87   4                  COMMAND$REGISTER = 0;
88   4                  BOX$PARAMS(I).ACCUM = BOX$PARAMS(I).ACCUM
                            + ADC$DATA$REGISTER;

89   4              END;
90   3              CALL RQSEND(.T$PARAM$LOCK,LOCK$MSG);
91   3              T$MSG = RQWAIT(.RET$TRIG,0);
92   3              CALL RQSEND(.TRIGGER,T$MSG);
93   3          END;

94   2      END ADC$TASK;
```

The CONTROL task waits for a message from the ADC task signaling that A/D readings have been taken and are ready for further processing. It completes the handshake by sending the message to the RET$TRIG exchange. Then, as in the ADC task, accesses the BOX$PARAMS data structure.

Inside the next loop, the readings are averaged, scaled, offset, and tested. Appropriate action is taken to turn the heating elements on or off. The loop concludes by returning the message to the T$PARAM$LOCK exchange.

```
95   1      CONTROL$TASK:
              PROCEDURE PUBLIC;
96   2          DECLARE (LOCK$MSG,T,MSG) ADDRESS;
97   2          DECLARE I BYTE;
98   2          DECLARE NCHNLS LITERALLY '5';
99   2          DECLARE TURN$LAMP$ON
                    LITERALLY 'OUTPUT(OE7H)=SHL(1,1)';
100  2          DECLARE TURN$LAMP$OFF
                    LITERALLY 'OUTPUT(OE7H)=SHL(I,1)+1';
101  2          DECLARE SETUP$8255 LITERALLY 'OUTPUT(OE7H)=80H;
                            OUTPUT(OE6H)=OFFH';
102  2          SETUP$8255;

104  2          DO FOREVER;
105  3              MSG = RQWAIT(.TRIGGER,0);
106  3              CALL RQSEND(.RET$TRIG,MSG);
107  3              LOCK$MSG= RQWAIT(.T$PARAM$LOCK,0);
108  3              DO I = 0 TO NCHNLS-1;
109  4                  BOX$PARAMS(I).COUNT = BOX$PARAMS(I).COUNT + 1;
110  4                  IF BOX$PARAMS(I).COUNT
                            = BOX$PARAMS(I).SAMPLES THEN
111  4                  DO;
112  5                      T,
                            BOX$PARAMS(I).READING
                            = (BOX$PARAMS(I).ACCUM
                            /BOX$PARAMS(I).SAMPLES) / 38
                            + BOX$PARAMS(I).OFFSET;
113  5                      IF T <= BOX$PARAMS(I).SET$POINT
                            - BOX$PARAMS(I).ERROR THEN
114  5                          TURN$LAMP$ON;
                            ELSE
115  5                          IF T >= BOX$PARAMS(I).SET$POINT
                            + BOX$PARAMS(I).ERROR THEN
116  5                              TURN$LAMP$OFF;
                            BOX$PARAMS(I).ACCUM,
                            BOX$PARAMS(I).COUNT = 0;
118  5                  END;
119  4              END;
120  3              CALL RQSEND(.T$PARAM$LOCK,LOCK$MSG);
121  3          END;

122  2      END CONTROL$TASK;

123  1      END ATOD;
```

SUMMARY/CONCLUSIONS

The purpose of this application note is to introduce you to the Intel RMX/80, Real-Time Multitasking Executive. The general framework of RMX/80 was discussed, including the nucleus and extensions.

This application note described the steps involved in using RMX/80. Key emphasis has been placed on the need to fully define the tasks and exchanges in your application using graphic notation.

Applications have been presented to demonstrate task communication, synchronization, and mutual exclusion in a minimal terminal handler and an analog subsystem. The tasks responded to real-time asynchronous events such as USART and A/D interrupts.

RMX/80 represents a significant step in the sophistication of microcomputer software. Its ease of use, flexibility, and power should enable you to quickly implement real-time software for your applications.

```
1              MINIMAL$TERMINAL$HANDLER:

               DO;

2      1       DECLARE TRUE LITERALLY '0FFH';
3      1       DECLARE FOREVER LITERALLY 'WHILE TRUE';

               /* SPECIAL ASCII CHARACTERS */
4      1       DECLARE
                  BELL          LITERALLY '07H',
                  LF            LITERALLY '0AH',
                  CR            LITERALLY '0DH',
                  CONTROL$R     LITERALLY '12H',
                  CONTROL$X     LITERALLY '18H',
                  ESC           LITERALLY '1BH',
                  RUBOUT        LITERALLY '7FH';

5      1       DECLARE EXCHANGE$DESCRIPTOR LITERALLY 'STRUCTURE (
                  MESSAGE$HEAD ADDRESS,
                  MESSAGE$TAIL ADDRESS,
                  TASK$HEAD ADDRESS,
                  TASK$TAIL ADDRESS,
                  EXCHANGE$LINK ADDRESS)';

6      1       DECLARE INT$EXCHANGE$DESCRIPTOR LITERALLY 'STRUCTURE (
                  MESSAGE$HEAD ADDRESS,
                  MESSAGE$TAIL ADDRESS,
                  TASK$HEAD ADDRESS,
                  TASK$TAIL ADDRESS,
                  EXCHANGE$LINK ADDRESS,
                  LINK ADDRESS,
                  LENGTH ADDRESS,
                  TYPE BYTE)';

7      1       DECLARE TH$MSG LITERALLY 'STRUCTURE(
                  LINK ADDRESS,
                  LENGTH ADDRESS,
                  TYPE BYTE,
                  HOME$EXCHANGE ADDRESS,
                  RESPONSE$EXCHANGE ADDRESS,
                  STATUS ADDRESS,
                  BUFFER$ADDRESS ADDRESS,
                  COUNT ADDRESS,
                  ACTUAL ADDRESS)';

               /*
                  8253 PORT ADDRESSES.
               */
8      1       DECLARE A8253$MODE LITERALLY '0DFH';
9      1       DECLARE A8253$CTR2 LITERALLY '0DEH';
```

```
          /*
            8253 COMMANDS.
          */
10    1   DECLARE SELECT$2 LITERALLY '10000000B';
11    1   DECLARE RL$BOTH  LITERALLY '00110000B';
12    1   DECLARE MODE$3 LITERALLY '00000110B';
13    1   DECLARE B2400 LITERALLY '001CH';


          /*
            8251 PORT ADDRESSES.
          */
14    1   DECLARE USART$IN LITERALLY '0ECH',
                  USART$OUT LITERALLY '0ECH',
                  USART$CONTROL LITERALLY '0EDH';


          /*
            8251 MODES.
          */
15    1   DECLARE STOP$1  LITERALLY '01000000B';
16    1   DECLARE CL8 LITERALLY '00001100B';
17    1   DECLARE RATE$16X LITERALLY '00000010B';


          /*
            8251 COMMANDS.
          */
18    1   DECLARE USART$RESET LITERALLY '01000000B',
                  RTS         LITERALLY '00100000B',
                  ERROR$RESET LITERALLY '00010000B',
                  RXE         LITERALLY '00000100B',
                  DTR         LITERALLY '00000010B',
                  TXEN        LITERALLY '00000001B';

19    1   RQSEND:
            PROCEDURE (EXCHANGE$POINTER,MESSAGE$POINTER) EXTERNAL;
20    2       DECLARE (EXCHANGE$POINTER,MESSAGE$POINTER) ADDRESS;
21    2     END RQSEND;

22    1   RQWAIT:
            PROCEDURE (EXCHANGE$POINTER,DELAY) ADDRESS EXTERNAL;
23    2       DECLARE (EXCHANGE$POINTER,DELAY) ADDRESS;
24    2     END RQWAIT;

25    1   RQELVL:
            PROCEDURE (LEVEL) EXTERNAL;
26    2       DECLARE LEVEL BYTE;
27    2     END RQELVL;

28    1   DECLARE RQINPX EXCHANGE$DESCRIPTOR PUBLIC;
29    1   DECLARE RQOUTX EXCHANGE$DESCRIPTOR PUBLIC;

30    1   DECLARE RQL6EX INT$EXCHANGE$DESCRIPTOR PUBLIC;
31    1   DECLARE RQL7EX INT$EXCHANGE$DESCRIPTOR PUBLIC;

32    1   INITIALIZATION:
            PROCEDURE;
33    2       OUTPUT(A8253$MODE) = SELECT$2 OR RL$BOTH OR MODE$3;
34    2       OUTPUT(A8253$CTR2) = LOW(B2400);
```

```
35  2          OUTPUT(A8253$CTR2) = HIGH(B2400);
36  2          OUTPUT(USART$CONTROL),
               OUTPUT(USART$CONTROL),
               OUTPUT(USART$CONTROL),
               OUTPUT(USART$CONTROL) = 0;
37  2          OUTPUT(USART$CONTROL) = USART$RESET;
38  2          OUTPUT(USART$CONTROL) = STOP$1 OR CL8 OR RATE$16X;
39  2          OUTPUT(USART$CONTROL) = RTS OR ERROR$RESET OR
                                       RXE OR DTR OR TXEN;
40  2       END INITIALIZATION;

41  1    RD$MIN:
            PROCEDURE PUBLIC;
42  2       DECLARE (MSGPTR,INTMSG,BUF$ADDRESS) ADDRESS;
43  2       DECLARE (CHAR,PTR,I) BYTE;
44  2       DECLARE MSG BASED MSGPTR TH$MSG;
45  2       DECLARE (BUF BASED BUF$ADDRESS) (1) BYTE;

46  2       ECHO$CHAR:
              PROCEDURE (CHAR);
47  3           DECLARE CHAR BYTE;
48  3           INTMSG = RQWAIT(.RQL7EX,0);
49  3           OUTPUT(USART$OUT) = CHAR;
50  3         END ECHO$CHAR;

51  2       CALL INITIALIZATION;

52  2       CALL RQELVL(6);
53  2       CALL RQELVL(7);

54  2       DO FOREVER;
55  3         MSGPTR = RQWAIT(.RQINPX,0);
56  3         BUF$ADDRESS = MSG.BUFFER$ADDRESS - 1;
57  3         PTR = 0;
58  3         CHAR = NOT CR;
59  3         DO WHILE CHAR <> CR;
60  4           INTMSG = RQWAIT(.RQL6EX,0);
61  4           IF (CHAR := INPUT(USART$IN) AND 7FH) = RUBOUT THEN
62  4           DO;
63  5             IF PTR = 0 THEN
64  5               CALL ECHO$CHAR(BELL);
                  ELSE
65  5             DO;
66  6               CALL ECHO$CHAR(BUF(PTR));
67  6               PTR = PTR - 1;
68  6             END;
69  5           END;
               ELSE
70  4           DO;
71  5             IF CHAR = CONTROL$X THEN
72  5             DO;
73  6               CALL ECHO$CHAR('#');
74  6               CALL ECHO$CHAR(CR);
75  6               CALL ECHO$CHAR(LF);
76  6               PTR = 0;
77  6             END;
                 ELSE
78  5             DO;
```

```
79   6              IF CHAR = CONTROL$R THEN
80   6              DO;
81   7                 CALL ECHO$CHAR(CR);
82   7                 CALL ECHO$CHAR(LF);
83   7                 DO I = 1 TO PTR;
84   8                    CALL ECHO$CHAR(BUF(I));
85   8                 END;
86   7              END;
                     ELSE
87   6              DO;
88   7                 IF PTR < MSG.COUNT THEN
89   7                    BUF(PTR := PTR+1) = CHAR;
                         ELSE
90   7                 DO;
91   8                    IF CHAR <> CR THEN
92   8                       CHAR = BELL;
93   8                 END;
94   7                 IF CHAR = ESC THEN
95   7                 DO;
96   8                    CALL ECHO$CHAR('$');
97   8                    CHAR = CR;
98   8                 END;
99   7                 CALL ECHO$CHAR(CHAR);
100  7              END;
101  6           END;
102  5        END;
103  4     END;
104  3     IF PTR < MSG.COUNT THEN
105  3        BUF(PTR:=PTR+1) = LF;
106  3     MSG.ACTUAL = PTR;
107  3     MSG.STATUS = 0;
108  3     CALL RQSEND(MSG.RESPONSE$EXCHANGE,MSGPTR);
109  3     CALL ECHO$CHAR(LF);
110  3  END;
111  2  END RD$MIN;

112  1  WR$MIN:
           PROCEDURE PUBLIC;
113  2     DECLARE (MSGPTR,INTMSG,BUF$ADDRESS) ADDRESS;
114  2     DECLARE PTR BYTE;
115  2     DECLARE MSG BASED MSGPTR TH$MSG;
116  2     DECLARE (BUF BASED BUF$ADDRESS) (1) BYTE;

117  2     CALL RQELVL(7);

118  2     DO FOREVER;
119  3        MSGPTR = RQWAIT(.RQOUTX,0);
120  3        BUF$ADDRESS = MSG.BUFFER$ADDRESS - 1;
121  3        DO PTR = 1 TO MSG.COUNT;
122  4           INTMSG = RQWAIT(.RQL7EX,0);
123  4           OUTPUT(USART$OUT) = BUF(PTR);
124  4        END;
125  3        MSG.ACTUAL = MSG.COUNT;
126  3        MSG.STATUS = 0;
127  3        CALL RQSEND(MSG.RESPONSE$EXCHANGE,MSGPTR);
128  3     END;
129  2  END WR$MIN;
130  1  END MINIMAL$TERMINAL$HANDLER;
```

WRMIN ASSEMBLY LANGUAGE LISTING

```
LOC    OBJ          SEQ              SOURCE STATEMENT

                      1              NAME     WRMIN
                      2              EXTRN    RQELVL,RQOUTX,RQWAIT,RQSEND
                      3              PUBLIC   WRMIN,RQL7EX
00EC                  4    DATOUT    EQU      0ECH     ; USART OUTPUT PORT ADR
                      5              CSEG
                      6    WRMIN:
0000 0E07             7              MVI      C,7
0002 CD0000    E      8              CALL     RQELVL   ; ENABLE INTERRUPT LVL 7
                      9    WR0:
0005 110000           10             LXI      D,0
0008 010000    E      11             LXI      B,RQOUTX
000B CD0000    E      12             CALL     RQWAIT   ; WAIT FOR OUTPUT RQST
000E E5               13             PUSH     H        ; PUSH MESSAGE ADDRESS
000F 110700           14             LXI      D,7
0012 19               15             DAD      D
0013 4E               16             MOV      C,M      ; GET RESPONSE EXCHANGE
0014 23               17             INX      H
0015 46               18             MOV      B,M
0016 23               19             INX      H
0017 C5               20             PUSH     B        ; PUSH RESPONSE EXCHANGE
0018 3600             21             MVI      M,0      ; STATUS = 0
001A 23               22             INX      H
001B 3600             23             MVI      M,0
001D 23               24             INX      H
001E 5E               25             MOV      E,M      ; GET BUFFER ADR′IN DE
001F 23               26             INX      H
0020 56               27             MOV      D,M
0021 23               28             INX      H
0022 4E               29             MOV      C,M      ; GET COUNT IN BC
0023 23               30             INX      H
0024 46               31             MOV      B,M
0025 23               32             INX      H
0026 71               33             MOV      M,C      ; ACTUAL = COUNT
0027 23               34             INX      H
0028 70               35             MOV      M,B
                      36   WR1:
0029 78               37             MOV      A,B
002A B1               38             ORA      C
002B CA4300    C      39             JZ       WR2      ; EXIT LOOP IF COUNT = 0
002E C5               40             PUSH     B
002F D5               41             PUSH     D
0030 110000           42             LXI      D,0
0033 010000    D      43             LXI      B,RQL7EX
0036 CD0000    E      44             CALL     RQWAIT   ; WAIT FOR TXRDY INTRPT
0039 D1               45             POP      D
003A C1               46             POP      B
003B 1A               47             LDAX     D
003C 13               48             INX      D
003D D3EC             49             OUT      DATOUT   ; TRANSMIT NEXT CHAR
003F 0B               50             DCX      B
0040 C32900    C      51             JMP      WR1
                      52   WR2:
```

```
0043 C1          53          POP     B       ; BC = RESPONSE EXCHANGE
0044 D1          54          POP     D       ; DE = MSG ADDRESS
0045 CD0000   E  55          CALL    RQSEND  ; SEND MSG TO RESP. EXCH
0048 C30500   C  56          JMP     WR0
                 57 ;
                 58          DSEG
                 59 RQL7EX:
000F             60          DS      15
                 61 ;
                 62          END
```

ATOD PL/M LISTING

```
1               ATOD:
                DO;

                $INCLUDE(:F1:COMMON.ELT)
2    1    =     DECLARE TRUE LITERALLY 'OFFH';
3    1    =     DECLARE FALSE LITERALLY '00H';
4    1    =     DECLARE BOOLEAN LITERALLY 'BYTE';
5    1    =     DECLARE FOREVER LITERALLY 'WHILE 1';

                $INCLUDE(:F1:EXCH.ELT)
6    1    =     DECLARE EXCHANGE$DESCRIPTOR LITERALLY 'STRUCTURE (
          =       MESSAGE$HEAD ADDRESS,
          =       MESSAGE$TAIL ADDRESS,
          =       TASK$HEAD ADDRESS,
          =       TASK$TAIL ADDRESS,
          =       EXCHANGE$LINK ADDRESS)';

                $INCLUDE(:F1:IED.ELT)
7    1    =     DECLARE INT$EXCHANGE$DESCRIPTOR LITERALLY 'STRUCTURE (
          =       MESSAGE$HEAD ADDRESS,
          =       MESSAGE$TAIL ADDRESS,
          =       TASK$HEAD ADDRESS,
          =       TASK$TAIL ADDRESS,
          =       EXCHANGE$LINK ADDRESS,
          =       LINK ADDRESS,
          =       LENGTH ADDRESS,
          =       TYPE BYTE)';

                $INCLUDE(:F1:MSG.ELT)
8    1    =     DECLARE MSG$HDR LITERALLY '
          =       LINK ADDRESS,
          =       LENGTH ADDRESS,
          =       TYPE BYTE,
          =       HOME$EXCHANGE ADDRESS,
          =       RESPONSE$EXCHANGE ADDRESS';
          =
9    1    =     DECLARE MSG$DESCRIPTOR LITERALLY 'STRUCTURE(
          =       MSG$HDR,
          =       REMAINDER(1) BYTE)';

                $INCLUDE(:F1:INTRPT.EXT)
10   1    =     RQENDI:
          =         PROCEDURE EXTERNAL;
          =
11   2    =         END RQENDI;
          =
12   1    =     RQELVL:
          =         PROCEDURE (LEVEL) EXTERNAL;
13   2    =           DECLARE LEVEL BYTE;
          =
14   2    =         END RQELVL;
```

```
15   1   =   RQDLVL:
         =     PROCEDURE (LEVEL) EXTERNAL;
16   2   =       DECLARE LEVEL BYTE;
         =
17   2   =     END RQDLVL;
         =
18   1   =   RQSETV:
         =     PROCEDURE (PROC,LEVEL) EXTERNAL;
19   2   =       DECLARE PROC ADDRESS;
20   2   =       DECLARE LEVEL BYTE;
         =
21   2   =     END RQSETV;

             $INCLUDE(:F1:SYNCH.EXT)
22   1   =   RQSEND:
         =     PROCEDURE (EXCHANGE$POINTER,MESSAGE$POINTER) EXTERNAL;
23   2   =       DECLARE (EXCHANGE$POINTER,MESSAGE$POINTER) ADDRESS;
         =
24   2   =     END RQSEND;
         =
25   1   =   RQWAIT:
         =     PROCEDURE (EXCHANGE$POINTER,DELAY) ADDRESS EXTERNAL;
26   2   =       DECLARE (EXCHANGE$POINTER,DELAY) ADDRESS;
         =
27   2   =     END RQWAIT;
         =
28   1   =   RQACPT:
         =     PROCEDURE (EXCHANGE$POINTER) ADDRESS EXTERNAL;
29   2   =       DECLARE EXCHANGE$POINTER ADDRESS;
         =
30   2   =     END RQACPT;
         =
31   1   =   RQISND:
         =     PROCEDURE (IED$PTR) EXTERNAL;
32   2   =       DECLARE IED$PTR ADDRESS;
         =
33   2   =     END RQISND;


             /*
                 SBC 711 ANALOG TO DIGITAL BOARD
             */
34   1       DECLARE ADC$BASE ADDRESS AT (0F700H);
35   1       DECLARE COMMAND$REGISTER BYTE AT (.ADC$BASE+0);
36   1       DECLARE STATUS$REGISTER BYTE AT (.ADC$BASE+0);
37   1       DECLARE FIRST$CHANNEL$REGISTER BYTE AT (.ADC$BASE+1);
38   1       DECLARE LAST$CHANNEL$REGISTER BYTE AT (.ADC$BASE+2);
39   1       DECLARE CLEAR$INTERRUPT$REQUEST BYTE AT (.ADC$BASE+3);
40   1       DECLARE ADC$DATA$REGISTER ADDRESS AT (.ADC$BASE+4);

41   1       DECLARE GO$BIT LITERALLY '1';
42   1       DECLARE AUTO$INCREMENT$ENABLE LITERALLY '2';
43   1       DECLARE BUSY LITERALLY '8';
44   1       DECLARE EOS$INTERRUPT$ENABLE LITERALLY '10H';
45   1       DECLARE EOC$INTERRUPT$ENABLE LITERALLY '20H';
46   1       DECLARE END$OF$SCAN LITERALLY '40H';
```

```
47    1          DECLARE END$OF$CONVERSION LITERALLY '80H';

48    1          DECLARE DUMMY EXCHANGE$DESCRIPTOR PUBLIC;
49    1          DECLARE RET$PULSE EXCHANGE$DESCRIPTOR PUBLIC;
50    1          DECLARE GO$PULSE EXCHANGE$DESCRIPTOR PUBLIC;
51    1          DECLARE TRIGGER EXCHANGE$DESCRIPTOR PUBLIC;
52    1          DECLARE RET$TRIG EXCHANGE$DESCRIPTOR PUBLIC;

53    1          DECLARE RQL2EX INT$EXCHANGE$DESCRIPTOR;

54    1          DECLARE T$PARAM$LOCK EXCHANGE$DESCRIPTOR EXTERNAL;

55    1          DECLARE BOX$PARAMS(5) STRUCTURE(
                         CHANNEL BYTE,
                         SET$POINT ADDRESS,
                         ERROR ADDRESS,
                         OFFSET ADDRESS,
                         SAMPLES ADDRESS,
                         COUNT ADDRESS,
                         ACCUM ADDRESS,
                         READING ADDRESS ) EXTERNAL;

56    1          TICKER$TASK:
                   PROCEDURE PUBLIC;
57    2             DECLARE MSG ADDRESS;
58    2             DECLARE PULSE$MSG(2) STRUCTURE (
                           MSG$HDR );

59    2             PULSE$MSG(0).LENGTH,
                    PULSE$MSG(1).LENGTH = SIZE(PULSE$MSG(0));
60    2             PULSE$MSG(0).TYPE,
                    PULSE$MSG(1).TYPE = 65;
61    2             CALL RQSEND(.RET$PULSE,.PULSE$MSG(0));
62    2             CALL RQSEND(.RET$PULSE,.PULSE$MSG(1));

63    2             DO FOREVER;
64    3                MSG = RQWAIT(.DUMMY,5);
65    3                MSG = RQWAIT(.RET$PULSE,0);
66    3                CALL RQSEND(.GO$PULSE,MSG);
67    3             END;

68    2             END TICKER$TASK;

69    1          ADC$TASK:
                   PROCEDURE PUBLIC;
70    2             DECLARE TRIGGER$MSG STRUCTURE (
                           MSG$HDR );
71    2             DECLARE (T$MSG,MSG,LOCK$MSG) ADDRESS;
72    2             DECLARE I BYTE;
73    2             DECLARE GAIN LITERALLY '00';
74    2             DECLARE N$CHNLS LITERALLY '5';

75    2             TRIGGER$MSG.LENGTH = SIZE(TRIGGER$MSG);
76    2             TRIGGER$MSG.TYPE = 65;
77    2             CALL RQSEND(.RET$TRIG,.TRIGGER$MSG);
78    2             CALL RQELVL(2);
```

```
79    2            DO FOREVER;
80    3               MSG = RQWAIT(.GO$PULSE,0);
81    3               CALL RQSEND(.RET$PULSE,MSG);
82    3               LOCK$MSG = RQWAIT(.T$PARAM$LOCK,0);
83    3               DO I = 0 TO N$CHNLS-1;
84    4                  FIRST$CHANNEL$REGISTER = BOX$PARAMS(I).CHANNEL
                                                + ROL(GAIN,6);
85    4                  COMMAND$REGISTER = GO$BIT
                                          OR EOC$INTERRUPT$ENABLE;
86    4                  MSG = RQWAIT(.RQL2EX,0);
87    4                  COMMAND$REGISTER = 0;
88    4                  BOX$PARAMS(I).ACCUM = BOX$PARAMS(I).ACCUM
                                            + ADC$DATA$REGISTER;
89    4               END;
90    3               CALL RQSEND(.T$PARAM$LOCK,LOCK$MSG);
91    3               T$MSG = RQWAIT(.RET$TRIG,0);
92    3               CALL RQSEND(.TRIGGER,T$MSG);
93    3            END;

94    2         END ADC$TASK;

95    1         CONTROL$TASK:
               PROCEDURE PUBLIC;
96    2            DECLARE (LOCK$MSG,T,MSG) ADDRESS;
97    2            DECLARE I BYTE;
98    2            DECLARE NCHNLS LITERALLY '5';
99    2            DECLARE TURN$LAMP$ON
                     LITERALLY 'OUTPUT(0E7H)=SHL(I,1)';
100   2            DECLARE TURN$LAMP$OFF
                     LITERALLY 'OUTPUT(0E7H)=SHL(I,1)+1';
101   2            DECLARE SETUP$8255 LITERALLY 'OUTPUT(0E7H)=80H;
                                               OUTPUT(0E6H)=0FFH';

102   2            SETUP$8255;

104   2            DO FOREVER;
105   3               MSG = RQWAIT(.TRIGGER,0);
106   3               CALL RQSEND(.RET$TRIG,MSG);
107   3               LOCK$MSG= RQWAIT(.T$PARAM$LOCK,0);
108   3               DO I = 0 TO NCHNLS-1;
109   4                  BOX$PARAMS(I).COUNT = BOX$PARAMS(I).COUNT + 1;
110   4                  IF BOX$PARAMS(I).COUNT
                          = BOX$PARAMS(I).SAMPLES THEN
111   4                  DO;
112   5                     T,
                            BOX$PARAMS(I).READING
                              = (BOX$PARAMS(I).ACCUM
                                /BOX$PARAMS(I).SAMPLES) / 38
                              + BOX$PARAMS(I).OFFSET;
113   5                  IF T <= BOX$PARAMS(I).SET$POINT
                                 - BOX$PARAMS(I).ERROR THEN
114   5                     TURN$LAMP$ON;
                         ELSE
115   5                     IF T >= BOX$PARAMS(I).SET$POINT
                                 + BOX$PARAMS(I).ERROR THEN
```

```
116   5                        TURN$LAMP$OFF;
                          BOX$PARAMS(I).ACCUM,
                          BOX$PARAMS(I).COUNT = 0;
118   5                END;
119   4              END;
120   3              CALL  RQSEND(.T$PARAM$LOCK,LOCK$MSG);
121   3            END;

122   2         END CONTROL$TASK;

123   1       END ATOD;
```

November 1978

Using FORTRAN-80 for iSBC™ Applications

Steve Verleye

OEM Microcomputer Systems Applications

9800836A

Using FORTRAN-80 for iSBC™ Applications

Contents

I. INTRODUCTION

In March of 1978, Intel announced the availability of a resident FORTRAN compiler for the Intellec® Microcomputer Development System. In November of 1978, Intel announced the availability of a run-time package to support the execution of FORTRAN-80 compiled programs in the RMX/80™ environment. With this support package, user's of Intel's complete line of iSBC™ Single Board Computer products can benefit from the full set of I/O and math capabilities provided by the FORTRAN-80 language.

This application note is intended to familiarize the reader with the features, benefits and usage of the FORTRAN-80 package and RMX/80™ Executive. The reader who is unfamiliar with any of these topics is urged to refer to the related Intel publications listed in the front-piece.

Following the overview, two application examples will be studied. In the first example, FORTRAN code is used in a "stand-alone" environment; i.e., without operating system support. The second example is a multitasking system managed by the RMX/80 Executive which supports standard I/O interfaces to the RMX/80 Terminal Handler and Disk File System.

II. OVERVIEW

Intel's FORTRAN-80 compiler is an implementation of the standard FORTRAN known as ANS FORTRAN 77 approved by the American National Standards Institute (ANSI) in April, 1978. The implementation is of the FORTRAN 77 subset, plus most of the full I/O capability and Intel defined extensions. For a fuller description

of the implementation, consult the *FORTRAN-80 Programming Manual.*

FORTRAN-80 is a high level applications programming language with flexible I/O handling and floating-point math instructions. With the FORTRAN-80 language, the programmer can easily implement sophisticated applications involving scientific calculations, process and instrument control, test and measurement, and a host of other applications requiring the power and flexibility the FORTRAN-80 language provides.

With the addition of the iSBC 801 FORTRAN-80 RUN-TIME PACKAGE for RMX/80 SYSTEMS, the user who wishes to implement his application using Intel's Single Board Computers and the RMX/80 Real-Time Multitasking Executive can take full advantage of the FORTRAN-80 I/O and math capabilities. The package allows the user to accelerate the run-time execution of FORTRAN-80 coded mathematical formulae through special interfaces to the optional iSBC 310™ High Speed Mathematics Unit. All disk and terminal I/O is interfaced directly to the RMX/80 Disk File System and either the full or the minimal Terminal Handler. The libraries that comprise the iSBC package are constructed in a modular fashion, allowing the user to configure systems with as much or as little of the support libraries as needed for a given application.

In order to effectively utilize the hardware and software products now available, it is important to design the application system from the top down. This implies that we need to think of an application in very general terms and then successively introduce more detail until we have program code as our final step. At each stage of the definition, we have to make decisions about the usage and configuration of various products.
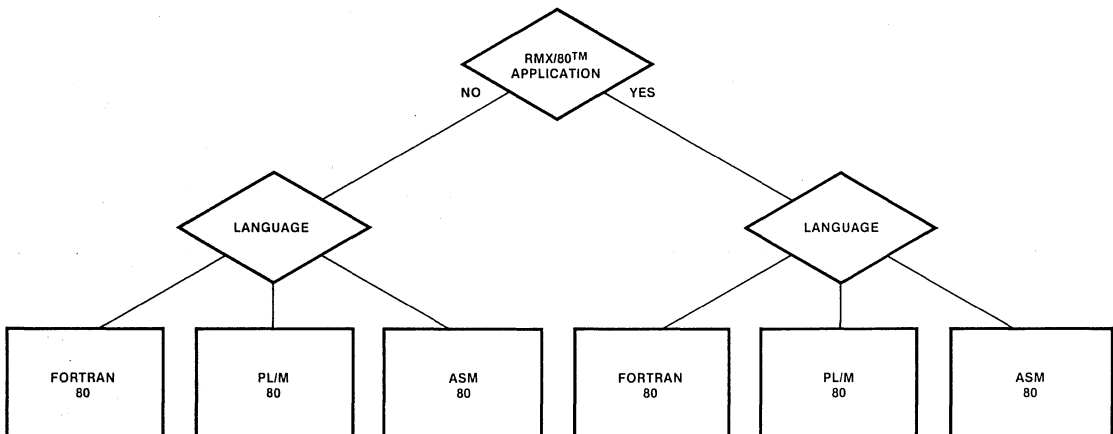
The decision-making process that concerns itself with software can be shown as a tree (Figure 1). The first decision that must be made is whether or not the RMX/80 Real-Time Multitasking Executive should be utilized. In general, this package will prove extremely useful if the application to be designed must respond to multiple asynchronous events, or contains multiple, semi-independent processes that could be executed in parallel, or has need of standard vendor supplied device drivers. If the application is very small and simple, handles few or no interrupts, has no need for parallel execution of multiple processes, and the designer is willing to supply his own I/O device drivers, the program may be able to execute without the support of an operating system.

Whether the RMX/80 package is used or not, the system designer must now choose in which language or languages the programs should be coded. Each of the three languages shown is optimized for different purposes. The PL/M-80 language is well suited for systems programming. The ASM-80 language is best suited for applications requiring direct control of the computer (e.g., the registers and memory). The FORTRAN-80 language is highly desirable for those applications requiring mathematical calculations and formatted

I/O. In many cases, the optimal solution will use a mix of two or even all three of these languages.

III. USING FORTRAN-80

I/O Capabilities

After the decision has been made to use the FORTRAN-80 language for an application, various types of I/O support are available to the user (see Figure 2). If the program code is to run without any support from an operating system, the user must supply drivers for any devices he wishes to include in his system.

When designing an RMX/80 system, the iSBC 801 package supplies the standard interface to the disk and terminal while the user may support additional devices in the same manner as the "stand alone" program would. The following sections expand on the topic of FORTRAN-80 I/O support.

Port I/O

The simplest and most direct method of performing I/O in the FORTRAN-80 language uses two pre-defined subroutines, INPUT and OUTPUT. The example below illustrates the use of these subroutines to input bytes from and output bytes to any of the 8080A/8085A I/O ports.

Figure 2. The I/O Support Decision

INTEGER* 1 IVAL

```
C
C -- PROGRAM THE 8255 PARALLEL I/O CHIP
C -- PORT # = EB; VALUE = 94
C
      CALL OUTPUT (#0EBH, # 94H)
    •
    •
    •
C
C -- INPUT 8 BITS FROM PORT A INTO IVAL
C -- PORT # = E8; VALUE  INPUT TO IVAL
C
      CALL INPUT (#0E8H,IVAL)
    •
    •
    •
```

Port I/O is extremely useful in many applications. However, this method requires the programmer to directly handle each and every byte. Also, it does not utilize the power of the FORTRAN-80 formatting routines.

Internal Buffer Formatting

One method of overcoming the shortcomings of Port I/O is to use character strings as "virtual devices." This is accomplished by specifying a character string as the unit number in a READ or WRITE statement. External routines can be used to fill the character strings with input for the READ statement and to output buffers that have been formatted by the WRITE statement. The example below shows the use of this method.

```
      SUBROUTINE EXAMPL
      CHARACTER*80 BUFFER
    •
    •
    •
C
C -- CALL DEVICE DRIVER TO GET BUFFER OF
C -- CHARACTERS
C
      CALL BUFIN (BUFFER)
C
C -- NOW READ FROM BUFFER INTO VARIABLES
C -- UNDER FORMAT CONTROL
C
      READ (BUFFER, 100) X, Y, Z
100 FORMAT (F10.3, F12.4, F13.5)
    •
    • PROCESS DATA STORED IN VARIABLES
    • X, Y, Z
    •
C
C -- WRITE RESULTS TO BUFFER
C
      WRITE (BUFFER, 200) A, B, C, D
200 FORMAT (4F12.3)
C
```

```
C -- CALL DEVICE DRIVER TO OUTPUT BUFFER
C
      CALL BUFOUT (BUFFER)
    •
    •
    •
```

User Provided High-Level Drivers

If an application requires only simple input (READ) and output (WRITE) capabilities, the previous method would probably be sufficient. If, however, the device(s) in the system are more complex, it may be necessary to perform other I/O operations. One way of doing this would be to write subroutines for each operation. A much nicer solution is to use the FORTRAN-80 I/O instructions (OPEN, CLOSE, READ, WRITE, PRINT, BACKSPACE, REWIND, and ENDFILE) to interface to user-written routines which implement these instructions for the special device.

This is possible because, for each open file in the system, the FORTRAN-80 I/O system keeps a table connecting the unit number with the addresses of the routines that handle all operations on that unit. The I/O system allows the user to substitute his own device drivers into this table. To do this, the system designer codes a routine and labels it FQ0LVL. This routine is then made known to the I/O system (i.e., declared PUBLIC). Whenever a file is first accessed (i.e., OPENED), the I/O system calls FQ0LVL with a set of parameters, one of which is the file name referenced in the OPEN statement. The designer, in his code for FQ0LVL, scans the file name to decide if this is one of the files for which he wishes to supply drivers. If so, he passes back a table of the addresses of the routines that will take care of the eight primitive file I/O capabilities (refer to the example following this paragraph and to the *FORTRAN-80 Compiler Operators Manual*).

```
FQ0LVL: PROCEDURE(file$ptr,buf$ptr) BYTE PUBLIC;

/* table of entry point addresses for driver routines */

DECLARE table (8) ADDRESS DATA(
    .open$hdlr,    /* address of OPEN routine */
    .close$hdlr,   /* address of CLOSE routine */
    .read$hdlr,    /* address of READ routine */
    .write$hdlr,   /* address of WRITE routine */
    .back$hdlr,    /* address of BACKSPACE routine */
    .mv2rec$hdlr,  /* address of MV2REC routine */
    .rewind$hdlr,  /* address of REWIND routine */
    .make$eof$hdlr /* address of END OF FILE routine */
    );

DECLARE (returned$status,index) BYTE;
DECLARE (file$ptr,buf$ptr) ADDRESS;
DECLARE buf BASED buf$ptr (1) BYTE;
DECLARE file$name BASED file$ptr (1) BYTE;
DECLARE analog$in (*) BYTE DATA(':AI:');

/* set flag initially =FFH */

returned$status=0FFH;

/* if any character of file$name does not compare set flag=0 */

    DO index=0 TO 3;
    IF file$name(index) <> analog$in(index) THEN
        returned$status=0;
    END;

/* if flag=FFH pass back the addresses of the drivers */

    IF returned$status=0FFH THEN
        CALL move(size(table),.table,buf$ptr);
    RETURN returned$status;
    END; /* of FQ0LVL */
```

RMX/80™ Support

When using the RMX/80 Executive, the iSBC 801 FORTRAN-80 RUN-TIME PACKAGE for RMX/80 SYSTEMS can be used to provide a direct interface to standard RMX/80 high level drivers, the Disk File System and the Terminal Handler. With the RMX/80 Executive, users can code multiple, concurrently executing programs that perform formatted I/O to disk files and the console, as shown in the following example:

```
C
C -- OPEN disk file
C
    OPEN (8,FILE = ':D0:TSTDTA.FIL',ACCESS =
    'SEQUENTIAL')
C
C -- perform tests
C
    •
    •
    •
C
C -- WRITE results to file for archival storage
C
    WRITE(8,100) (RESULT (I),I = 1,10)
100 FORMAT(10F12.3)
C
C -- PRINT completion message on console
C
    PRINT 200
200 FORMAT ('TESTS COMPLETE')
    •
    •
    •
```

If it is necessary for a FORTRAN program in the RMX/80 system to perform I/O to a device not handled by one of the high level drivers, any of the methods previously described can be utilized to augment the I/O system.

FORTRAN-80 Math Capabilities

The FORTRAN-80 language supports four data types labelled INTEGER, REAL, LOGICAL, and CHARACTER. Also supported are various operators which can manipulate objects of various types. Both INTEGER (fixed point) and REAL (floating-point) objects can be manipulated by the add (+), subtract (−), multiply (*), divide(/), and exponentiation (**) operators. In addition, Integers can be operated on by the Boolean operators (e.g., .AND.,.OR.). In this case, the operations are performed bit-wise on the operands.

All floating-point arithmetic operations are performed with algorithms that adhere to the Intel Floating-Point Standard[1] which allows for seven decimal digits of precision. Whenever math operations are used, the user

can make the decision to use a software package to implement the floating point support or to accelerate the execution of these operations (by as much as a factor of five or six) by installing an iSBC 310 High-Speed Mathematics Unit and linking in special FORTRAN-310 drivers. In either case, due to the adherence to the standard, the results of all calculations will be identical. In addition, the libraries have been designed to allow the switch to be made from software routines to a faster hardware solution with *no* code changes.

Above and beyond the basic mathematical operators in FORTRAN-80, a large number of intrinsic functions are available. These functions provide services like type conversion, remaindering, and logarithmic and trigonometric calculation. Since the calculations involved in performing these high-level functions require the mathematical operators, they too can be accelerated by the inclusion of the iSBC 310 board and its associated drivers.

Error Handling

The math processing system also provides flexible error handling. The user can choose to use either an Intel-supplied error handler or one of his own design. The capability also exists to change the active error handler dynamically in cases where different routines require different handlers. The default error handlers are named FQFERH. One exists in each of the arithmetic libraries (Figure 3). This error handler will attempt to recover from an error by taking the most reasonable action (e. g., underflow error returns result = 0). If code is being run "stand-alone" or under the RMX/80 executive the handlers in the math libraries should be used or the user should supply his own. Appendix B of the *ISIS-II FORTRAN-80 Compiler Operator's Manual* contains all of the information necessary to implement a custom error handler or to use the default routines.

FPSOFT.LIB	- Software package for "stand-alone" and ISIS-II systems
FPHARD.LIB	- iSBC 310 drivers for same
*FPSFTX.LIB	- Software package for RMX/80 systems
*FPHRDX.LIB	- iSBC 310 drivers for iSBC 80/20, 80/20-4 and 80/30 boards
*FPHX10.LIB	- iSBC 310 drivers for iSBC 80/10 and 80/10A boards
FPEF.LIB	- Library of routines implementing intrinsic functions

*Available in iSBC 801 FORTRAN-80 RUN-TIME PACKAGE for RMX/80 Systems.

Figure 3. Available Math Libraries

---

1 Palmer, John F., "The Intel Standard for Floating-Point Arithmetic," *Proceedings of the First International Computer Software and Applications Conference* (Chicago: IEEE Computer Society), November, 1977, pp 107-112.

IV. APPLICATION EXAMPLE

An Automated Test Stand

This example shows the steps taken to design and implement an automated test stand. The hardware system must interface to a test fixture upon which test items can be mounted. Operator inputs and test outputs involve a 300-baud hard copy terminal. The software to be developed must allow an operator to invoke a variety of tests from the console and to receive some printed performance record for the object under test. In addition, the software must allow for tests to be added and deleted often, and each test must be allowed to obtain any number of parameters from the command line tail.

After examining the problem definition and the decision making diagram presented earlier, it was decided that this application could be implemented with a simple sequential program.

Since formatted I/O and mathematical calculations are involved, the FORTRAN-80 language is well suited to be the main programming language. Also, some ASM-80 routines will come in handy for communicating with the console.

An analysis of the I/O to be performed breaks down into two distinct types. Various inputs to and outputs from the text fixture will be 8-bit parallel transfers. These will likely go through the 8255A ports on the Single Board Computer. Port I/O will be used to handle this function. Interface with the operator requires READ'S AND WRITE's to the console device. The simplest way of performing this function is to use character strings as the target of READ and WRITE operations and coding small ASM-80 routines to transfer these buffers from/to the console.

A diagram of the test stand is shown in Figure 4. The computer hardware necessary to solve this application includes a Single Board Computer (the iSBC 80/20 board), a PROM memory module and an analog I/O board. Digital I/O with the test fixture is handled by the 8255A ports on the Single Board Computer. The analog inputs on the test fixture come from the two D/A converter channels on the iSBC 732 board.

The software solution utilizes a very rudimentary command line interpreter. The mainline routine gets a line of input and finds the first non-blank character. If this character is an alphabetic character, it is used in a computed GOTO statement to transfer control to one of a possible 26 entry points. Tests may be added by choosing a keyletter and inserting a label in the GOTO statement to transfer control to the new test routine. The command input line and the index in the line are stored in a common block so that any test routine can continue scanning the line for parameters or can reset the index and find out what keyletter caused its invocation. The flow of the software is illustrated in Figure 5.

For the purpose of explanation, routines are shown to implement a "calculator mode" which allows the opera-

tor to perform arithmetic from the console, and a logic transition tester which determines whether the object on the test fixture changes state at the proper voltages.
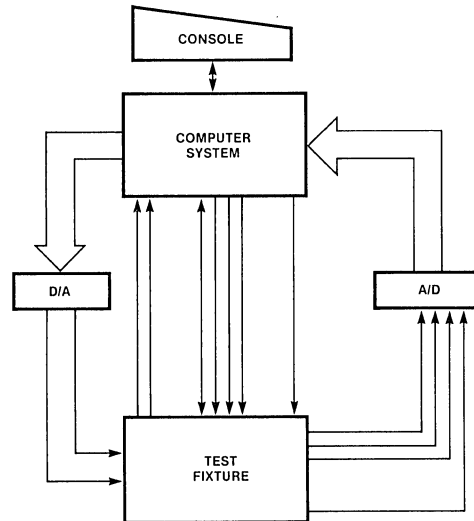
Figure 4. Test Stand Diagram

Code Description

The following sections describe the program code for this application example. Fold-out code listings are contained in Appendix A. The circled reference letters in the text refer to the corresponding letters in the listings.

The DRIVRS Module

The module DRIVRS contains three primary routines. START Ⓐ is located at 0 so that it is executed upon power up. This routine is responsible for programming the on-board hardware (8255A, 8251, 8253), setting up the system stack, and calling the FORTRAN routine labeled MAINLN.

The input routine BUFIN Ⓑ is called from FORTRAN routines with a character string as an argument. Note that passing a string argument from FORTRAN results in the address and length of the string being sent as parameters. The string is filled with characters input from the console until a carriage return is encountered. A simple line-editing scheme is implemented allowing character deletion (RUBOUT), line deletion (CONTROL-X), and echoing of the current buffer contents (CONTROL-R). Attempted entry of characters beyond the end of the string and RUBOUTS past the beginning cause the audible bell to sound.

The output routine, BUFOUT Ⓒ , also takes a character string as an argument. The entire contents of the string are sent to the console unless a carriage return is encountered in the string. If a carriage return is the terminator, a line feed is output as well. If a CONTROL-S is entered at the console while output is in progress, output is suspended until a CONTROL-Q is typed.
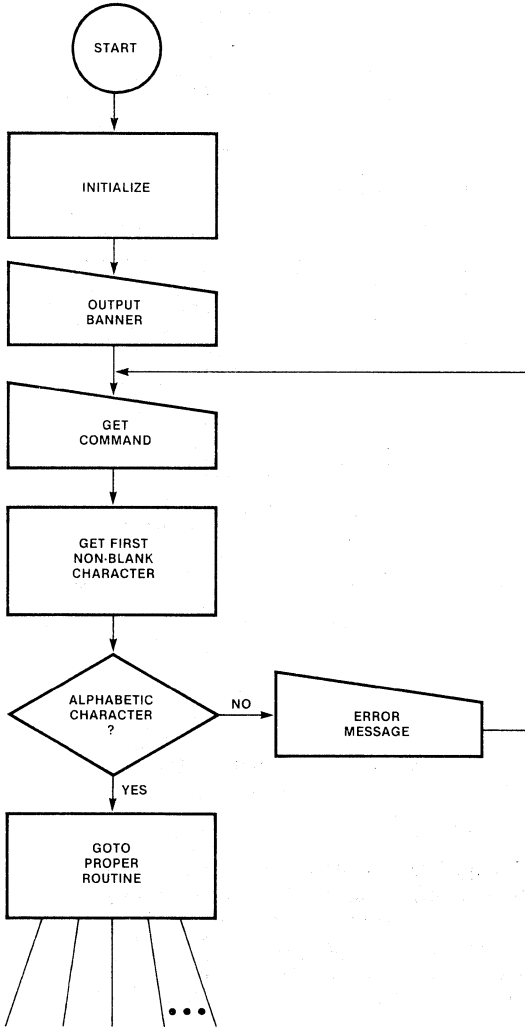
Figure 5. Flow Diagram

The MAINLN Module

The module MAINLN Ⓓ contains the mainline routine that implements the command line interpreter. The statement IMPLICIT LOGICAL A-Z will cause most usages of undeclared variables to be reported as illegal mixed mode; the intent in writing these programs was to declare all variables, which is generally considered

good programming practice, even though Fortran makes default assumptions about undeclared variables.

The default handler is to be used for any errors that may occur while performing mathematical calculations. Also, the routines that perform the calculations must be initialized. Both of these operations are performed by the call to FQFSET Ⓔ . The call takes two arguments. The first argument is a two byte field specifying which error handler is to be used. If the low order bit of the high order byte is a one (e.g., 100 hexadecimal), the math routines will call a user error handler whose address is given as the second parameter. If the low order bit is zero (as is the case in this example), the routines will use the default handler and ignore the second argument.

A banner is output to the console by the sequence at Ⓕ where a formatted WRITE is performed on an internal buffer (IMAGE) and then the external driver BUFOUT is called to output the buffer to the console. The variable CARRET is used to insert a carriage return into the string to be output. In order to allow individual characters in the character string to be accessed, the EQUIVALENCE statement is used to cause LINBUF and IMAGE to occupy the same memory space. The variable INDEX Ⓖ is used to scan through the input buffer.

A call to BUFIN Ⓗ fetches the command line from the console. DBLANK is called Ⓘ to position INDEX to the first non-blank character. This character is converted to its integer representation, normalized to 1 and checked to see if it is a valid alphabetic character Ⓙ . If the keyletter is valid, the computed GOTO Ⓚ causes execution to branch to the correct point in a jump table Ⓛ . Note that A (add,) S (subtract), M (multiply), and D (divide) all branch to a single routine MATH, T (transition test) branches to a routine called TRANST and all other keyletters are trapped into line 100. Any and all I/O errors cause the ERROR routine to be called.

The DBLANK Module

The DBLANK routine Ⓜ de-blanks the input line. If a carriage return is encountered, the operator is prompted for more input.

The ERROR Module

The ERROR routine Ⓝ prints out an error message, with the error number, to the console.

The MATH Module

In many of the tests, the human operator must supply numeric parameters. A calculator mode is supplied for the simple calculations that might be needed here. This mode is implemented through the MATH routine Ⓞ . Since any one of four keyletters could have caused this routine to be invoked, MATH rescans the command line to obtain the keyletter Ⓟ . Following this, two operands are read in by calls to CONVRT Ⓠ and the operation requested is performed on them.

The CONVRT Module

Subroutine CONVRT (R) is called from other routines to extract floating-point operands from the input line buffer. Characters are transferred into a temporary buffer (S) until either a carriage return or a comma is encountered. The temporary buffer is then read under format control to obtain the returned value (T) .

The TRANST Module

The item to be tested is composed of combinatorial logic as shown in Figure 6. The transition test sets all inputs except one to a constant value. By varying the voltage at the remaining input, the transitions at the output can be checked. This test must be run while the +5V power to the fixture is varied through a range of values. This testing is performed by the TRANST routine.

Power is supplied to the test fixture through one of the two D/A channels on the iSBC™ 732. Three of the input parameters specify the starting and stopping voltage values for Vcc and the increment to be added each step. The fourth parameter is the tolerance to be used to decide if the test passes or fails at each step. Once the test is running, the output voltage at (2) is measured for inputs at (1) of 0 and 5V. The voltage input is then incremented from 0V (using D/A channel 1) until a transition is sensed in the output voltage at (2) . At this point, the input voltage at (1) is checked to see if it is within tolerance. The same process is then repeated with the voltage at (1) going from +5V downward. After the test is complete, a formatted report is generated containing the ambient temperature (measured through a temperature sensor) and the performance record for the item under test.
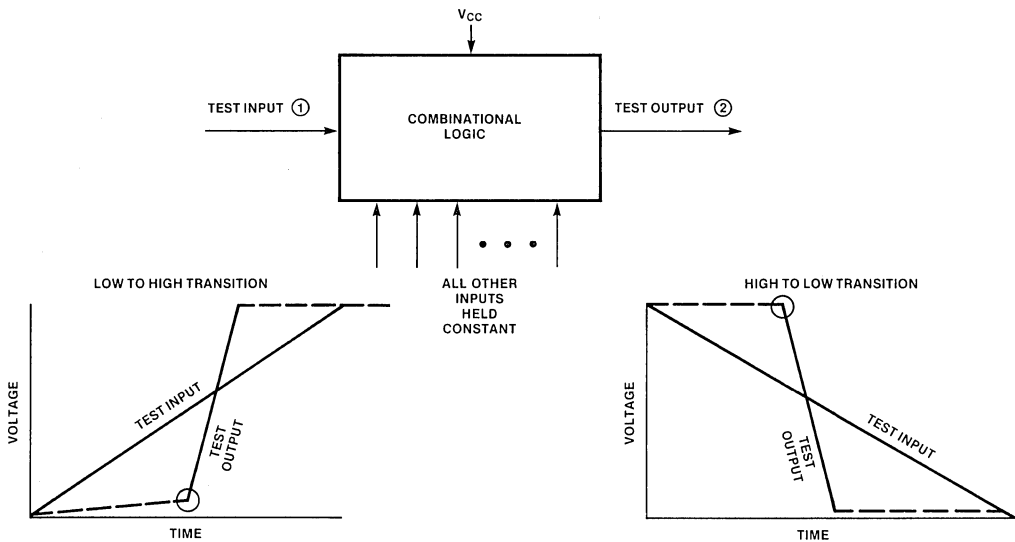
Figure 6. Transition Test

```
TEST STAND V0.0
COMMAND?
M 34.678,345.43
         34.67800 *        345.43000 =          11978.82160
COMMAND?
T 4.5,5.5,.2,5.
TRANSITION TEST   TOLERANCE=  5.0% AMBIENT TEMPERATURE = 25.30 DEGREES C
     VCC    HIGH TRANS   LOW TRANS   HIGH    LOW     TEST
     4.5       00.81       3.42      4.43    0.12    PASSED
     4.7       00.80       3.44      4.67    0.08    PASSED
     4.9       00.80       3.67      4.88    0.02    PASSED
     5.1       00.80       3.71      4.93    0.02    PASSED
     5.3       00.80       3.73      4.98    0.01    PASSED
     5.5       00.80       3.74      4.99    0.01    PASSED
COMMAND?
```

Figure 7. Sample Output

An external routine Ⓤ , ADCIN, is called to input samples into the variable given as the first parameter from the channel given as the second parameter. The counts from the temperature sensor exhibit a logarithmic curve, so the input is linearized using the equation shown. The routine DACOUT Ⓥ takes the first parameter and outputs it to the channel specified by the second parameter. If no transition occurs when the test input is run through its entire range, the item is assumed non-functional, a message is output, and control is returned to the console Ⓦ .

The ADCIN Module

Subroutine ADCIN Ⓧ fetches samples from the A/D converter on the iSBC 732 board. The channel number is an input parameter and the data is the returned value. Of special note in this routine is the use of the FORTRAN common block to control a memory-mapped device. The master CPU communicates with the iSBC 732 by way of memory read and write commands instead of I/O commands. The primary reason for this is the fact that the 8080A IN and OUT instructions operate on only 8 bits at a time whereas SHLD and LHLD instructions can manipulate 16 bit operands. This is convenient when working with 12-bit inputs from the A/D and 12 bit outputs to the D/A. In the code, a common block is created which has the same makeup as the memory mapped registers on the iSBC 732 board. The common block will be origined at the address of the iSBC 732 by the ISIS-II LOCATE program.

The DACOUT Module

Subroutine DACOUT Ⓨ makes use of the same common block to output given values to a specified D/A channel.

LINK and LOCATE

The ISIS-II LINK command needed to pull together the individual pieces of this example is shown in Figure 8. After compilation, the object modules of all of the previously described routines are placed in the library FRTMOD.LIB by the ISIS-II Library Manager™. The LINK statement starts with the module DRIVRS.OBJ, which has one EXTERNAL reference, MAINLN. To satisfy this reference, MAINLN.OBJ is linked in from FRTMOD.LIB and its EXTERNAL references cause the inclusion of other modules.

The LOCATE command shown in Figure 9 is used to assign absolute memory locations to the code in the LINKED modules. The common block labelled ADC is explicitly assigned to FFF0H so that it will correctly overlay the memory-mapped space of the iSBC 732 board. The ORDER statement is used to tell the locator in what order the various segments should be placed in memory.

```
LINK :Fl:DRIVRS.OBJ, &
     :Fl:FRTMOD.LIB, &
     :F0:F80RUN.LIB, &
     :F0:F80NIO.LIB, &
     :F0:F80ISS.LIB, &
     :F0:FPEF.LIB,   &
     :F0:FPSOFT.LIB, &
     :F0:PLM80.LIB   &
TO :Fl:TSTND.LK0 PRINT(:Fl:TSTND.LNK) MAP
```

Figure 8. LINK Command for Test Stand Example

V. USING THE FORTRAN-80 RUN-TIME PACKAGE FOR RMX/80™ SYSTEMS

The iSBC 801 package provides I/O interface and math routines for users who are coding RMX/80 applications in the FORTRAN-80 language. In the following sections, an overview of the RMX/80 system will be presented along with a discussion of the use of the iSBC 801 package. This overview is not intended to be exhaustive. If the reader is unfamiliar with the RMX/80 package, he should gain from this section enough understanding to comprehend the concepts in the example presented. If the reader is planning on implementing an RMX/80 system, the RMX/80 references in the frontpiece should be studied carefully.

```
LOCATE :Fl:TSTND.LK0 PRINT(:Fl:TSTND.LOC) MAP LINES SYMBOLS PUBLICS &
ORDER(CODE DATA /LINE/ /ADC/) /ADC/(0FFF0H) STACKSIZE(0) CODE(0)
```

Figure 9. Locate Command for Test Stand Example

Overview of the RMX/80™ Executive

A large number of microcomputer applications require the ability to respond to events in real-time. The RMX/80 Executive provides the system software around which you can build a real-time multitasking application using Intel iSBC 80™ Single Board Computers. In addition, the RMX/80 package provides the application designer with various high-level drivers (such as a terminal handler and a disk file system) which make it easier to develop sophisticated applications software.

The RMX/80™ Model

At this time, it is appropriate to discuss the RMX/80 model, or in other words, the general concepts upon which the RMX/80 Executive is built. Real-time systems, such as the RMX/80 system, provide the capability to control and respond to events occurring asynchronously in the physical world. To handle these events, the application is broken up into smaller semi-independent pieces, and each of these pieces is brought into action to handle the event for which it is intended. Each of these independent program units is a *task*. The RMX/80 Executive manages the execution of these tasks in accordance with a user-designated priority scheme to insure that the highest-priority task in the system has control of the CPU. It is also necessary, in a system such as this, for these semi-independent program units (tasks) to communicate with each other. This communication may be for the purpose of synchronization, data passing, mutual exclusion or any other use that may arise. To facilitate inter-task communication, the RMX/80 model incorporates the notion of messages and exchanges. A *message* is a data structure that can contain an arbitrary amount of information to be communicated from one task to another. An *exchange* is a "mail box" where tasks may send messages to be picked up by other tasks. The primary operations (primitives) that accomplish message transfers in the RMX/80 system are RQSEND* and RQWAIT*. Figure 10 diagramatically shows the interaction of tasks, messages, and exchanges and introduces the symbolism used to represent these RMX/80 concepts in the system design.

Tasks

Typically, a task will execute a section of code that performs some initialization and then enters an infinite loop performing some processing over and over again as shown in Figure 11.

Each task in the system has a priority associated with it. The RMX/80 Executive uses this priority scheme to determine which ready task to run. The assignment of priorities to individual tasks is up to the system designer, giving him the capability to tune his system by assuring timely execution of important functions.
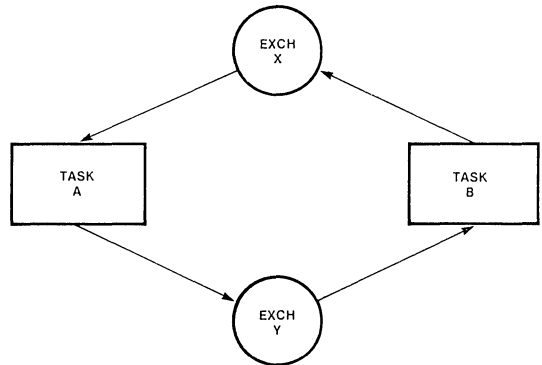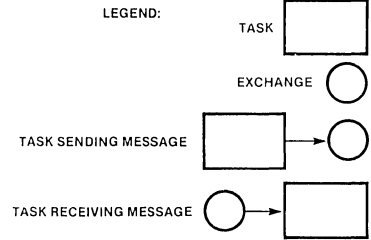
Figure 10. Task, Message, Exchange Interaction

```
      SUBROUTINE TASK 1
C
C -- DECLARATION OF VARIABLES HERE
C
C
C -- INITIALIZE VARIABLES AND I/O PORTS
C
      CALL OUTPUT (#0E8H,0)
      FLAG = 1
      INDEX = 1
      COUNT = 0
      SUM = 0
C
C -- ENTER INFINITE LOOP
C
1     CALL INPUT(#0E9H,IVAL)
      •
      ○
      •
      GOTO 1
      END
```

Figure 11. Task Loop

*In order to differentiate RMX/80 procedures and data structures from the user's, the names of system objects are always preceded by RQ.

2-43

Each RMX/80 task also has its own stack, and there is no system stack. Whenever a task must give up the processor (e.g., must wait for the occurance of an interrupt) all of the information necessary to reawaken it at some future time without affecting the results of it's processing is stored on its stack.

Exchanges

An *exchange* in the RMX/80 system is a data structure that contains pointers to lists of tasks and messages. Whenever a message is sent to an exchange where there are no tasks waiting, it is added to the list of messages at that exchange until a task accepts it. Similarly, if a task waits at an exchange for a message and there is no message in the list, the task is added to the list of tasks waiting at that exchange. In both cases, the tasks and messages are serviced on a first come, first served basis. Figure 12 shows the possible states an exchange may be in at a given time.
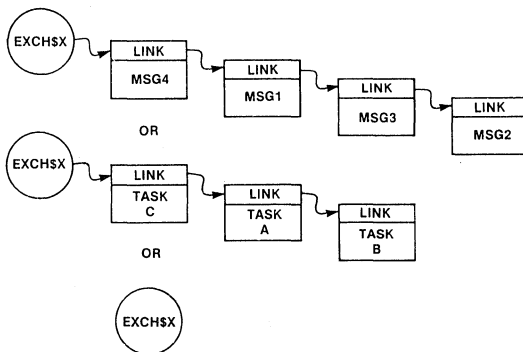
Figure 12. Exchange Lists

Messages

A *message* in the RMX/80 system is a contiguous section of memory of an arbitrary length. Information can be stored in the message prior to it being sent to an exchange where it will be accepted by another task.

Configuration

The configuration module contains various tables and PUBLIC variables that are accessed by the system at start-up time. All of the necessary information on the tasks and exchanges to be created and the disk file

system to be utilized are contained in this section of code. Configuration modules can be coded in either PL/M or assembly language (for which there are macros included with the RMX/80 product.)

Memory Usage

In systems using disk, it is necessary to ensure that certain buffers used by the disk controller for Direct Memory Access (DMA) are located in memory that is accessible to the disk controller. The buffers needed are allocated in a separate module called the *controller addressable memory* module. In the case of the iSBC 80/10, 80/10A, 80/20, and 80/20-4 boards, this module should be LOCATED before being included in the LINK statement to make sure that it does not contain any RAM on the CPU board itself (and, therefore, not controller-addressable). This restriction does not apply to iSBC 80/30 systems, since the iSBC 80/30 board has a dual port bus allowing system access to on-board RAM.

VI. APPLICATION EXAMPLE

A Sewage Treatment Plant Control System

In the early 1900's, the most popular type of sewage treatment system was known as a Sequencing Batch Reactor. It provided excellent effluent quality, but as populations grew, the amount of control necessary to operate the plant became too great for human operators, and a new type of treatment system came into use. This new system did not require such accurate control, but it also did not perform as well. With the passage of stricter and stricter water quality laws, and with the advent of low-cost, high powered microcomputer control systems, a serious look is being taken once again at Sequencing Batch Reactors.

A diagram of the treatment system and its sensors and actuators is shown in Figure 13. The system usually consists of three tanks, with each tank having individual influent and effluent valves, mixers and aeration equipment.

At any given time, all influent is being routed to one tank. When this tank is filled, the influent is routed to one of the other two. The full tank is agitated and aerated until the bacteria in the tank digest the sewage to within given limits. At this time, the mixer and aerator are turned off and the contents settle. After a time, the supernatant fluid is drawn off leaving the layer of concentrated bacteria to digest the next batch.

The computer control system necessary for controlling these reactors is shown in Figure 14. The system is responsible for monitoring the various sensors and contact closures, maintaining archives of system status, logging reports upon command, activating operator alarms, and performing on-line control of the batch cycle.
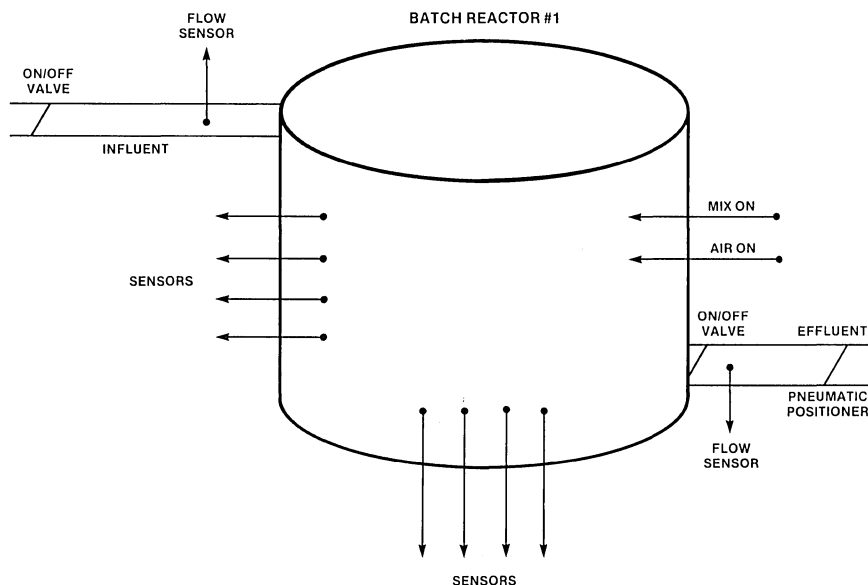
Figure 13. Sewage Treatment System and Sensors

Software

An analysis of the functions that need to be performed by the software for this control system leads to a decision to use the RMX/80 Executive. Timely response to multiple asynchronous events is the main thrust of this application. A breakdown of the individual functions in the system would be:

- *data collection* — gathers inputs from the sensors and contact closures and stores them where other routines can access them. Also, converts data from analog counts to engineering units.
- *on-line control* — based on current sensor inputs determines whether aeration, agitation, discharge and fill should be on or off.
- *alarm scanning* — compares current status values with setpoints and sets operator alarms if conditions are out of tolerance when effluent is on.
- *data logging* — once every five minutes logs current system status into a disk file record.
- *real-time clock* — maintains day, month, year, and time of day.
- *operator console handler* — monitors operator console to detect operator commands for time and setpoint changes, report generation, alarm clearing, etc.
- *report generation* — upon operator command, formats either the file corresponding to yesterday's operation or today's operation to the current moment.

Each of these functions must be studied independently

before the decision on which language to use for each is made. The functions concerned with data collection, on-line control, and alarm scanning will be concerned with mathematical calculations. The functions concerned with data logging and report generation will have need of formatted disk and console I/O. These routines will thus be coded in the FORTRAN-80 language.

As was mentioned earlier, the PL/M-80 language is a systems programming language. This means that it is optimized to deal with the concepts embodied in a high-level system such as the RMX/80 system. The program code that implements the real-time clock and operator console handler will be written in the PL/M-80 language. In addition, various PL/M-80 support routines will be written to be called on by one or more of the FORTRAN-80 routines. The purpose of these routines will be explained as they come up in the code descriptions following.

Hardware

The hardware used to implement this control system must perform the following functions:
- inputting analog samples from the various sensors
- outputting analog values to the pneumatic positioners
- inputting digital values from the contact closures and operator console
- outputting digital values to the operator console and alarm panel
- storing and retrieving data from diskette files

2-45
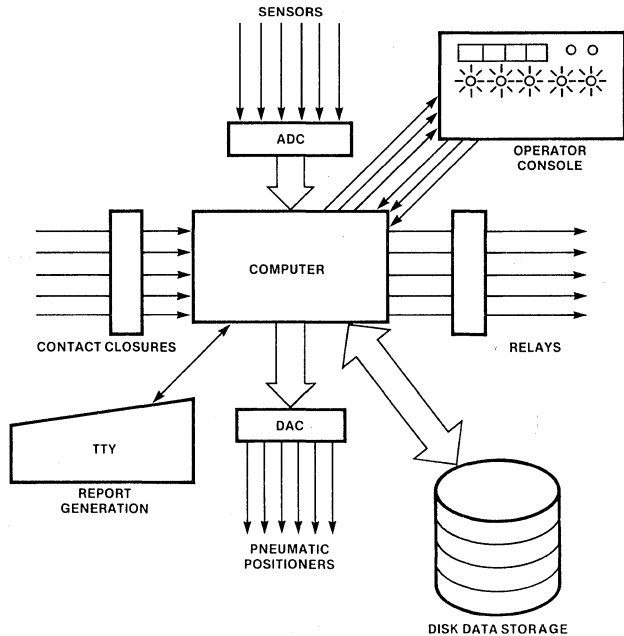
Figure 14. Computer Control System

The hardware configuration chosen includes an iSBC 80/30 Single Board Computer, an iSBC 732 Combination Analog Input/Output Board, a combination of PROM and RAM memory modules, and an iSBC 201 Diskette Controller.

There are various types of I/O devices in this system and each will require different FORTRAN-80 I/O support. The terminal and disk devices are supported through the iSBC 801 run-time package and the RMX/80 high level drivers. Communications with the A/D and D/A converters is accomplished using internal buffer formatting in conjunction with the RMX/80 Analog Handlers. Finally, port I/O is used for the digital inputs and outputs.

The next step in the design is to assign the system software functions to individual tasks in a manner that will allow for their parallel execution. The following tasks will be created to handle this application:

- STSINP — status input and unit conversion
- CNTROL — on-line control
- SCAN and TIMER5 — alarm scanning and data logging
- TIMER and TIMUPD — real-time clock
- CONSOL — operator console handler
- REPORT — report generation

Figure 15 shows the interaction of these tasks in the RMX/80 system.

System Considerations

At this point, let us consider some of the mechanisms this system will require to synchronize and co-ordinate the tasks we have created. Status and setpoint information will be stored in FORTRAN common blocks. This will allow the STSINP, CNTROL, SCAN, CONSOL, and TIMUPD tasks access to the STATUS information, and the CNTROL, SCAN, and CONSOL tasks access to the SETPNT information. Once per five minutes, SCAN will be notified through a flag byte (MIN5UP) that he is to write the current system status to the file TODAYS.RPT. Upon command from the operator, REPORT will need to read these files to generate reports.

Since the RMX/80 system is designed to handle asynchronous events, it is quite possible for any of the tasks to be pre-empted at any point in their execution (e.g., an interrupt occurs or a higher priority task becomes ready to run). Thus, the SCAN task may be in the process of reading the last byte of a four-byte REAL integer when STSINP pre-empts the SCAN task and writes new information into the STATUS common block, thus invalidating the current SCAN operation. In another instance, REPORT may be in the process of fetching a disk record when SCAN attempts to write to the file. For these reasons, and more, it is necessary to implement some sort of synchronization mechanism in this system. We will insure that at most, one task has access to the common blocks and disk records by using a technique called *mutual exclusion*. In the RMX/80 system, this is accom-
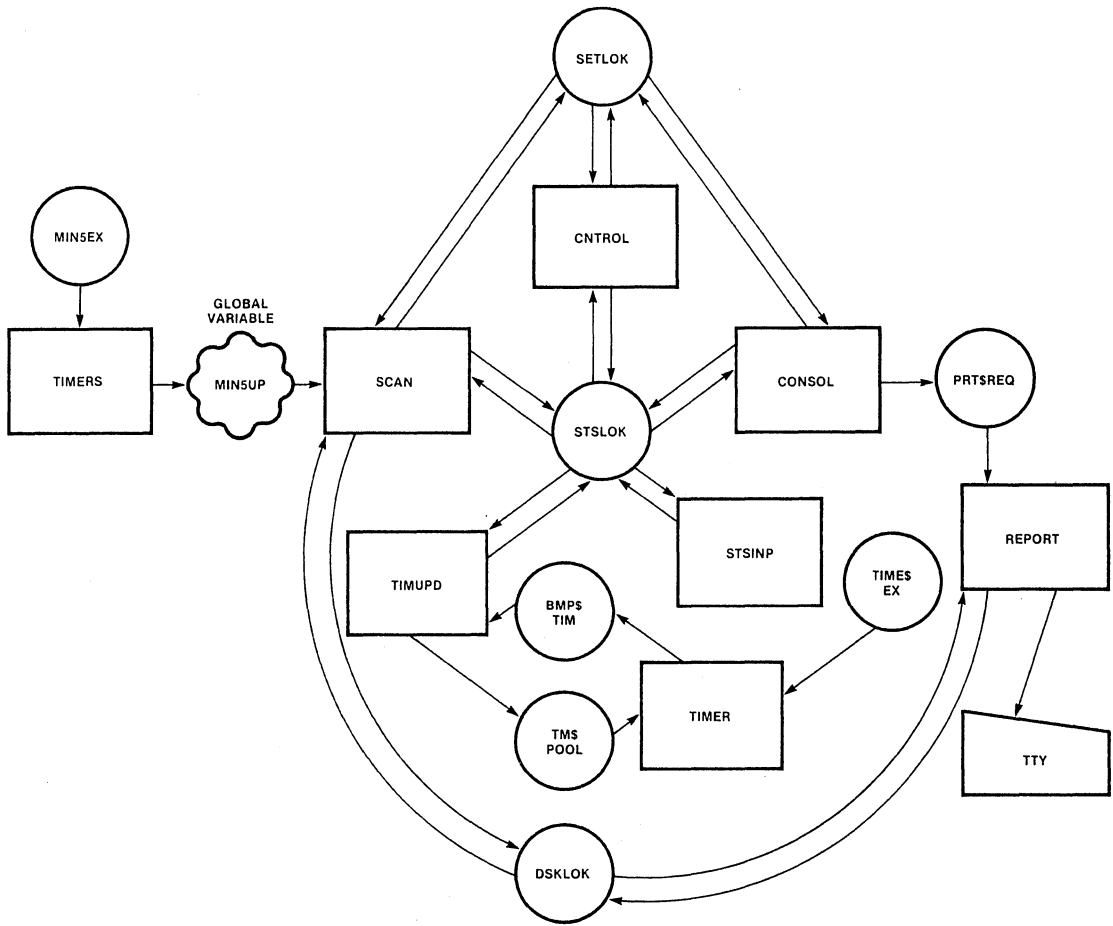
Figure 15. System Design Diagram

---

plished by creating an exchange for each shared resource and initially sending one message to it. Any task requiring access to the resource first waits at the associated exchange for the key message. If a message is at the exchange, the task obtains the message and continues running until finished and then sends the message back. If another task waits at the exchange while the first is processing, it will stop execution until the first task finishes and returns the message.

Code Descriptions

What follows is a description of the code used to implement most of the tasks discussed. Appendix B contains fold-out code listings with circled reference letters. In the description, sections of the code will be called out using

circled letters that correspond to symbols in the appendix.

The Semmod Module

Two PL/M routines called LOCK and UNLOCK perform the mutual exclusion operation discussed earlier. There are three exchanges used for the purpose of exclusion: STSLOK, SETLOK, and DSKLOK. They govern access to the STATUS common block, the SETPNT common block and the disk file respectively. The LOCK procedure Ⓐ takes one parameter, a number representing one of the three exchanges, and performs a wait at the appropriate exchange. Note the use of based variables to access the parameter. This is necessary since FORTRAN passes parameters by reference (address) rather than by value. The UNLOCK procedure Ⓑ takes the same parameter and sends the single key (message) back to the appropriate exchange.

These routines are written in the PL/M language because they must deal directly with a few system concepts that the FORTRAN-80 language does not. In particular, the RQWAIT routine returns to the caller the address of the message received from the exchange. In either the PL/M-80 or ASM-80 languages this address can be used to access the information in the message received. FORTRAN-80 routines do not have the capability to use address values to access data outside of their own module.

The STSINP Module

The module STSINP Ⓒ performs the function of updating the STATUS common block with new data from the sensors that has been converted to engineering units. STSINP initializes the FORTRAN-80 math routines Ⓓ and directs them to use the default error handler. STSINP then calls INIT$IO Ⓔ which initializes the message that will be used to communicate with the RMX/80 Analog Input Handler. The call to SMPLIN Ⓕ fills the buffer with analog samples from the sensors, and the following DO loop right-justifies the 12-bit samples in the 16 bit field Ⓖ . STSINP now waits for access to the STATUS block, converts the samples, stores them, inputs and stores the values of the contact closures and calls UNLOCK Ⓗ . The function performed by STSINP is not a continuous function. Update of the status information once per second is sufficient. The call to WAIT Ⓘ delays execution for one second.
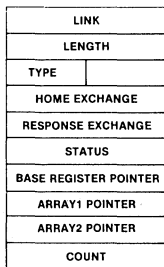
Figure 16. Request Message for Sequential Channel Input with Single Gain

The ANALOGIOMOD Module

In the module labelled ANALOGIOMOD, the declaration of READ$MSG Ⓙ uses the predefined LITERAL called AI$MSG. This LITERAL is one of many in the RMX/80 package that can be used to attach meaningful symbolic names to PL/M data structures. In this instance, AI$MSG defines the fields of a standard analog input request message. Figure 16 is a diagram of the individual

fields of the request message. The definition and usage of each of these fields is described in the *RMX/80 User's Guide*. The procedure INIT$IO Ⓚ is called by STSINP. It simply initializes the analog input request message and returns. Note the assignment operation in line 29. The RESPONSE$EXCHANGE field of the request message must contain the address of the exchange where the RMX/80 Analog Handler should send the response to the request (see Figure 17 for a diagram of the request-response mechanism). In the PL/M language, this address is assigned using a location reference - a variable name preceded by a period, which stands for the address of the variable. FORTRAN-80 routines lack the ability to refer to the address of variables.
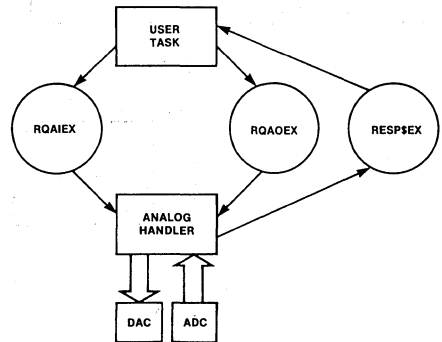
Figure 17. Request/Response Mechanism

The procedure SMPLIN Ⓛ fills in a buffer, given as an input parameter, with analog samples from sequential channels on the A/D. Note the mechanism used to handle the passing of a FORTRAN string as a parameter. For every string in the parameter list, FORTRAN passes the starting address of the string followed by its length in bytes.
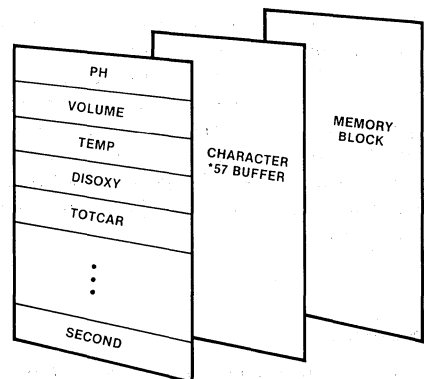
Figure 18. Use of EQUIVALENCE Statement

The SCAN Module

The SCAN task is responsible for operator alarms and data logging. The EQUIVALENCE statements (M) cause the STATUS common block to be overlaid by a character string, as illustrated in Figure 18. This allows for a compact file on disk of numerical data which can be broken out later for report generation.

After initialization, SCAN waits for access to both the STATUS and SETPNT common blocks. Operator alarms need to be set only if a parameter is out of specification and the effluent pump is on (N). After performing the scan, the variable MIN5UP is checked (O) to see if a report should be logged. If so, SCAN gains access to the disk file and writes a single record (P). All locks are now released, a one-second delay is counted out, and SCAN repeats the whole process. Normally, any errors that occur during the execution of I/O statements in the run-time package cause a message to be output on the console and the offending task to be suspended. Since this action is often undesirable, it is wise to handle one's errors programatically (Q).

The MIN5MOD Module

The module MIN5MOD contains two procedures. Both routines make use of the timed wait facility in the RMX/80 system. Any time a task calls RQWAIT to wait for a message at an exchange, an optional time limit (in 50 msec. intervals) can be specified. This is useful if the designer does not wish the task to be hung up forever if a message is never sent to that exchange. This mechanism can also be used to implement a timed wait if an exchange is specified to which no one will ever send a message. WAIT (R) delays execution of the calling task for one second. TIMER5 (S) waits for five minutes and then sets the variable MIN5UP to signal SCAN to log a disk record of current status.

The REPORT Module

The system console contains two buttons, one each for requests for printouts of today's and yesterday's status reports. Whenever one of these two buttons is pushed, the CONSOL task sends a message to the PRTREQ exchange with the TYPE field indicating which file to print. REPORT accepts these messages, checks the TYPE field (T), and calls the FORTRAN subroutine PRINT with the appropriate filename as a parameter. It then returns the request message to its sender via the response exchange field and waits for another request. Figure 19 is an example of the report generated by this system.

The PRINT Module

The PRINT subroutine will read in the compressed records written by SCAN and use the same set of EQUIVALENCE statements to break out the numerical data so

that it can be formatted for printout. If the type field (U) indicates that today's file is being accessed, PRINT obtains the key to the DSKLOK exchange since SCAN may disturb output operations if it attempts to log a new record. If yesterday's file is being accessed, the lock is not necessary, since no other task will be accessing this file. Once the lock is obtained, a record is read, the digital value of the contact closure status is converted to a more readable form (V) (ON or OFF), and the status line is formatted and printed. Since the SCAN task has an important function (operator alarms), we do not wish to hold it up for long if it happens to want to log a new status record. For this reason, PRINT relinquishes access to the file after every tenth record to allow SCAN to log its record and continue on. The rest of the code (W) checks for end of file indications and returns when printing is finished.

The INITMOD Module

Last in order (but first in execution) is the INIT procedure. It is called from the TIMER task, which is the highest-priority task in the system (and thus, will be the first to execute after start-up). INIT's role in life is to call FQ0GO (X) to initialize the FORTRAN I/O system, send one message to each of the lock exchanges (Y), and initialize the operator alarm panel (Z). The call to FQ0GO is a requirement for an RMX/80 system in which any FORTRAN-80 code that makes use of the iSBC 801 package is to be executed. The call must be made prior to the execution of any FORTRAN-80 I/O or math instructions. Also, the call to FQ0GO should only be made *once.*

DATE	TIME	PH	VOLUME	TEMP	DISSOLVED OXYGEN	TOTAL CARBON	ORGANIC CARBON
			(CU.M)	(C)	(MG/ML)	(MG/ML)	(MG/ML)
9/19/78	8: 5: 0	6.1	2012.32	25.4000	12.3400	76.9800	34.0070
9/19/78	8:10: 0	6.2	2614.08	25.4000	12.5400	88.0340	40.4933

SUSPENDED SOLIDS	PHOSPHATE CONC	INFLUENT FLOW	EFFLUENT FLOW	TURBID	AIR DIS	MIX	INF
(MG/ML)	(MG/ML)	(MG/ML)	(MG/ML)				
16.0987	56.9808	112.090	0.000	74.56	ON OFF	ON	ON
19.3943	61.4300	119.340	0.000	86.43	ON OFF	ON	ON

Figure 19. Sample Output

SYSTEM GENERATION

Configuration Module

Now that all of the code for the individual tasks is written, it is time to generate the tables that give the RMX/80 Executive the information it needs to configure all of the tasks and exchanges. Assembly language macros are included in the RMX/80 product to help make building the configuration module a little easier. After the counters have been initialized, the STD macro is invoked several times to define Static Task Descriptors for the tasks in the system. Of special note are the last two entries (AA). Any task that uses the FORTRAN I/O system must allocate approximately 800 bytes of stack. This extra stack space is needed to save information on the

current I/O operations. Also, any task performing floating point calculations with the software package needs to append an extra 18 bytes to its Task Descriptor as a workspace area. If the iSBC 310 drivers are used this need be only 13 bytes long. This last field is defined by passing a value of 13 or 18 to the optional parameter, TDXTRA, in the STD macro. The routines in the FORTRAN-80 Run-Time Package require one exchange, FQ0LOK, which is allocated using the XCH macro (BB), and added to the Initial Exchange Table by the PUBXCH macro (CC).

Controller Addressable Memory Module

The CAMMOD module (DD) allocates the blocks of memory needed by the RMX/80 Disk File System. Specific details on the contents of this module can be found in the *RMX/80 User's Guide*.

LINK

Figure 20 shows the ISIS-II LINK command used to bind all of the individual modules together with the RMX/80 libraries needed to implement this application. The FORTRAN-80 I/O interface routines are found in the library F80RMX.LIB which is part of the iSBC 801 package. the library FPSFTX.LIB contains the software floating point package. If it is desired to accelerate the execution of the mathematical operations in this system, the iSBC 310 board can be included and the library changed to FPHRDX.LIB for iSBC 80/20, 80/20-4, and 80/30 systems or FPHX10.LIB for iSBC 80/10 and 80/10A systems.

The RMX/80 extensions included are the Disk File System, the Analog Handlers, and the Minimal Terminal Handler.

LOCATE

After the Link has been finished, the command shown in Figure 21 is used to invoke the ISIS-II LOCATE program. The ORDER statement sets the proper order for all of the

different segments and common blocks. The common blocks themselves are allocated as fixed blocks of memory to make possible their shared usage by PL/M routines using the AT attribute. This mechanism is discussed in greater detail in AP-44, "How to use FORTRAN With Other Intel Languages".

VII. SUMMARY

The purpose of this application note has been to describe the design process used to decide what operating system support to use, what language to code programs in, what hardware to use and what type of I/O to use to solve a given application problem. The specific application examples presented have keyed on the use of the FORTRAN-80 language.

The lesson that has been learned is that proper design techniques result in the use of the right tool for every job. With a complete set of programming languages, each optimized for a specific use, a powerful real-time executive, and a complete line of flexible hardware products, complicated applications become easy to solve.

```
LINK :F0:RMX830.LIB(START), &
     :F1:X2CFG.OBJ, &
     :F1:RPTMOD.OBJ, &
     :F1:FRTMOD.LIB, &
     :F1:INITMD.OBJ, &
     :F0:F80RUN.LIB, &
     :F0:F80RMX.LIB, &
     :F0:FPEF.LIB, &
     :F0:FPSFTX.LIB, &
     :F0:SYSTEM.LIB, &
     :F0:DFSDIR.LIB(DIRECTORY,DELETE,RENAME,SEEK), &
     :F0:DIO830.LIB, &
     :F0:DFSUNR.LIB, &
     :F1:CAM.OBJ, &
     :F1:PLMMOD.LIB, &
     :F0:AIHDLR.LIB, &
     :F0:AOHDLR.LIB, &
     :F0:MTI830.LIB, &
     :F0:MTO830.LIB, &
     :F0:RMX830.LIB, &
     :F0:UNRSLV.LIB, &
     :F0:PLM80.LIB TO :F1:SEWAGE.LK0 PRINT(:F1:SEWAGE.LNK) MAP
```

Figure 20. LINK Command for Sewage Treatment Example

```
LOCATE :F1:SEWAGE.LK0 PRINT(:F1:SEWAGE.LOC) MAP &
    CODE(0) STACKSIZE(0) LINES SYMBOLS PUBLICS &
    ORDER(CODE DATA /LSTREC/ /STATUS/ /SETPNT/ /MIN5/) /STATUS/(FFA5H) &
    /SETPNT/(FFDEH) /MIN5/(FFEEH) /LSTREC/(FFA3H)
```

Figure 21. LOCATE Command for Sewage Treatment Example

APPENDIX A
CODE LISTINGS

```
  LOC  OBJ       SEQ         SOURCE STATEMENT

                   1 NAME     DRIVRS
                   2 ;
                   3 ;++++++++++++++++++++++++++++++++++++++++++++++++++++
                   4 ;
                   5 ;        CONSOLE I/O ROUTINES FOR FORTRAN-iSBC SYSTEM.
                   6 ;        START INITIALIZES THE HARDWARE AND CALLS THE
                   7 ;        FORTRAN ROUTINE MAINLN. BUFOUT ACCEPTS TWO
                   8 ;        PARAMETERS FROM THE CALLING FORTRAN ROUTINE
                   9 ;        (ACTUALLY ONE FROM THE ROUTINE SINCE PASSING
                  10 ;        A STRING ARGUMENT FROM FORTRAN RESULTS
                  11 ;        IN THE ADDRESS AND LENGTH OF THE STRING BEING
                  12 ;        SENT) AND OUTPUTS THE STRING TO THE USART
                  13 ;        ON THE 80/20.  BUFIN TAKES THE SAME TWO
                  14 ;        ARGUMENTS AND FILLS IN THE BUFFER WITH
                  15 ;        CHARACTERS UNTIL <CR> IS ENCOUNTERED. LINE
                  16 ;        EDITING IS PROVIDED TO THE EXTENT THAT
                  17 ;        RUBOUT DELETES A CHARACTER AND ECHOES IT,
                  18 ;        CONTROL-X DELETES THE BUFFER AND STARTS OVER,
                  19 ;        AND CONTROL-R PRINTS THE BUFFER CONTENTS.
                  20 ;        BUFOUT CALLS THE ROUTINE CHKIO TO DETERMINE
                  21 ;        IF A CNTL-S HAS BEEN ENTERED TO CAUSE A PAUSE
                  22 ;        IN THE OUTPUT. IF ENCOUNTERED THE ROUTINE
                  23 ;        WAITS UNTIL A MATCHING CNTL-Q IS ENTERED.
                  24 ;
                  25 ;++++++++++++++++++++++++++++++++++++++++++++++++++++
  000D            26 CR       EQU     0DH     ;ASCII CODE FOR CARRIAGE RET.
  000A            27 LF       EQU     0AH     ;ASCII CODE FOR LINE FEED
  001B            28 ESC      EQU     1BH     ;ASCII CODE FOR ESCAPE
  0018            29 CNTLX    EQU     18H     ;ASCII CODE FOR CONTROL-X
  007F            30 RUBOUT   EQU     07FH    ;ASCII CODE FOR RUBOUT
  0008            31 BS       EQU     08H     ;ASCII CODE FOR BACKSPACE
  0013            32 CNTLS    EQU     13H     ;ASCII CODE FOR CONTROL-S
  0011            33 CNTLQ    EQU     11H     ;ASCII CODE FOR CONTROL-Q
  0007            34 BELL     EQU     07H     ;ASCII CODE FOR BELL
  0012            35 CNTLR    EQU     12H     ;ASCII CODE FOR CONTROL-R
  00ED            36 CSTS     EQU     0EDH    ;USART COMMAND/STATUS PORT ADD
  00EC            37 CDATA    EQU     0ECH    ;USART DATA PORT ADDRESS
  0001            38 TXRDY    EQU     01H     ;MASK FOR TRANSMITTER READY
  0002            39 RXRDY    EQU     02H     ;MASK FOR RECEIVER READY
  0040            40 RESET    EQU     40H     ;USART RESET COMMAND
  004E            41 USMODE   EQU     4EH     ;USART MODE WORD
  0027            42 USCMND   EQU     27H     ;USART COMMAND WORD
  00B6            43 TIMCMD   EQU     0B6H    ;BAUD RATE CNTR COMMAND WORD
  0092            44 CMD55    EQU     92H     ;8255 COMMAND WORD
  00DE            45 CNTR2    EQU     0DEH    ;BAUD RATE CNTR PORT ADDRESS
  00DF            46 TIMCP    EQU     0DFH    ;TIMER CONTROL PORT ADDRESS
  00EB            47 PR8255   EQU     0EBH    ;8255 COMMAND PORT ADDRESS
  0080            48 STKSIZ   EQU     128     ;STACK SIZE
  00E0            49 BDFCTR   EQU     224     ;BAUD RATE FACTOR(COUNT VALUE)
                  50 ;
                  51 ;        ALLOCATE STACK
                  52 ;
```

```
  LOC  OBJ       SEQ         SOURCE STATEMENT

                  53           DSEG
  0080            54 FRTSTK: DS      STKSIZ
                  55 ;
                  56 ;        LOCAL DATA STORAGE
                  57 ;
  0002            58 BUFPTR: DS      2       ;BUFFER POINTER STORAGE
                  59           CSEG
                  60 ;
                  61 ;        START--STARTUP ROUTINE PROGRAMS THE USART
                  62 ;                AND TIMER THEN CALLS THE FORTRAN
                  63 ;                ROUTINE. IF THE FORTRAN ROUTINE
                  64 ;                RETURNS START SIMPLY STARTS OVER.
                  65 ;
                  66           EXTRN   MAINLN
  0000 317F00  D  67 START: LXI   SP,FRTSTK+STKSIZ-1 ;SET STACK POINTER
  0003 AF         68        XRA   A        ;ZERO ACCUMULATOR
  0004 D3ED       69        OUT   CSTS     ;BRING USART TO KNOWN STATE
  0006 D3ED       70        OUT   CSTS     ;BY SENDING FOUR NULLS
  0008 D3ED       71        OUT   CSTS     ;
  000A D3ED       72   (A)  OUT   CSTS     ;
  000C 3E40       73        MVI   A,RESET  ;RESET USART
  000E D3ED       74        OUT   CSTS     ;
  0010 3E4E       75        MVI   A,USMODE ;SEND USART MODE WORD
  0012 D3ED       76        OUT   CSTS     ;
  0014 3E27       77        MVI   A,USCMND ;SEND USART COMMAND
```

```
0016 D3ED        78        OUT     CSTS    ;
0018 3EB6        79        MVI     A,TIMCMD ;SEND COMMAND WORD
001A D3DF        8C        OUT     TIMCP   ;
001C 3EEC        81   (A)  MVI     A,LOW(BDFCTR) ;SEND LOW ORDER BYTE
001E D3DE        82        OUT     CNTR2   ;OF COUNTER VALUE
0020 3E00        83        MVI     A,HIGH(BDFCTR) ;SEND HIGH BYTE OF
0022 D3DE        84        OUT     CNTR2   ;COUNTER VALUE
0024 3E92        85        MVI     A,CMD55 ;8255 COMMAND WORD
0026 D3EB        86        OUT     PR8255  ;PROGRAM 8255
0028 CD0000   E  87        CALL    MAINLN  ;CALL FORTRAN ROUTINE
002B C30000   C  88        JMP     START   ;IF ROUTINE RETURNS START OVER
                 89  ;
                 90  ;        BUFIN--FILLS BUFFER WITH INPUT FROM TERMINAL
                 91  ;
                 92        PUBLIC  BUFIN
                 93  BUFIN:
002E E5          94        PUSH    H       ;SAVE HL PAIR
002F F5          95        PUSH    PSW     ;SAVE PSW
0030 C5          96        PUSH    B       ;SAVE BC
0031 60          97        MOV     H,B     ;GET BUFFER POINTER TO HL
0032 69          98        MOV     L,C     ;
0033 228000   D  99        SHLD    BUFPTR  ;SAVE IT
0036 1600        100       MVI     D,0     ;ZERO TO # CHARACTERS COUNTER
                 101                       ;NOTE: STRING LENGTH <=255
0038 D5          102       PUSH    D       ;SAVE COUNTERS
                 103 GETCHR:
0039 CDE200   C  104       CALL    CI      ;GET CHARACTER
003C FE7F        105       CPI     RUBOUT  ;RUBOUT?
003E C24700   C  106       JNZ     BUF05   ;NO,CONTINUE
0041 CD9000   C  107       CALL    DLTCHR  ;YES,DELETE LAST CHARACTER


LOC   OBJ       SEQ  (B)   SOURCE STATEMENT

0044 C33900   C  108       JMP     GETCHR  ;GET NEW ONE
                 109 BUF05:
0047 FE18        110       CPI     CNTLX   ;CONTROL-X?
0049 CAA200   C  111       JZ      DLTLIN  ;YES,DELETE BUFFER
004C FE12        112       CPI     CNTLR   ;CONTROL-R?
004E CAF900   C  113       JZ      PRTBUF  ;YES,PRINT BUFFER
0051 1D          114       DCR     E       ;DECREMENT SPACE LEFT COUNTER
0052 C26300   C  115       JNZ     BUF10   ;CONTINUE IF COUNTER > 0
0055 FE0D        116       CPI     CR      ;IF THIS END OF LINE ALL IS OK
0057 CA6300   C  117       JZ      BUF10   ;
005A 1C          118       INR     E       ;BRING COUNTER BACK ONE
005B 0E07        119       MVI     C,BELL  ;NOT OK, ECHO BELL
005D CDB400   C  120       CALL    ECHO    ;
0060 C33900   C  121       JMP     GETCHR  ;GET NEW CHARACTER
                 122 BUF10:
0063 4F          123       MOV     C,A     ;MOVE CHARACTER TO C
0064 CDB400   C  124       CALL    ECHO    ;AND ECHO IT
0067 71          125       MOV     M,C     ;STORE IT IN BUFFER
0068 23          126       INX     H       ;INCREMENT BUFFER POINTER
0069 14          127       INR     D       ;INCREMENT # OF CHARS COUNTER
006A 3E0D        128       MVI     A,CR    ;IS IT A NEWLINE CHARACTER
006C B9          129       CMP     C       ;
006D C23900   C  130       JNZ     GETCHR  ;NO,CONTINUE FILLING
0070 D1          131       POP     D       ;YES,RETURN
0071 C1          132       POP     B       ;
0072 F1          133       POP     PSW     ;
0073 E1          134       POP     H       ;
0074 C9          135       RET             ;RETURN
                 136 ;
                 137 ;        BUFOUT ENTRY POINT
                 138 ;
                 139       PUBLIC  BUFOUT
0075 E5          140 BUFOUT: PUSH  H       ;SAVE HL REGISTER PAIR
0076 F5          141       PUSH    PSW     ;SAVE PSW
0077 60          142       MOV     H,B     ;GET STRING POINTER INTO HL
0078 69          143       MOV     L,C     ;
0079 CDCD00   C  144 OUTCHR: CALL  CHKIO   ;CHECK FOR PAUSE(CNTL-S)
007C 4E          145       MOV     C,M     ;GET CHARACTER
007D CDB400   C  146       CALL    ECHO    ;OUTPUT TO TERMINAL
0080 3E0D        147  (C)  MVI     A,CR    ;IS IT A <CR>?
0082 B9          148       CMP     C       ;
0083 CA8D00   C  149       JZ      EXITLP  ;YES,EXIT
0086 23          150       INX     H       ;INCREMENT POINTER
0087 1B          151       DCX     D       ;DECREMENT STRING COUNT
0088 7A          152       MOV     A,D     ;GET HI BYTE
0089 B3          153       ORA     E       ;AND WITH LO BYTE
008A C27900   C  154       JNZ     OUTCHR  ;IF STRING COUNT <> 0 CONTINUE
008D F1          155 EXITLP: POP    PSW     ;RESTORE PSW
008E E1          156       POP     H       ;RESTORE HL
008F C9          157       RET             ;ALL THROUGH
                 158 ;
                 159 ;        DLTCHR--DELETES LAST CHAR ENTERED INTO BUFFER
                 160 ;
                 161 DLTCHR:
0090 15          162       DCR     D       ;DECREMENT # OF CHARS COUNTER
```

```
LOC   OBJ        SEQ         SOURCE STATEMENT

0091 F29B00  C   163         JP      DLTC10   ;IF >=0 CONTINUE
0094 14          164         INR     D        ;RUBOUT PAST START OF BUFFER
0095 0E07        165         MVI     C,BELL   ;INCREMENT COUNT,ECHO A BELL
0097 CDB400  C   166         CALL    ECHO     ;
009A C9          167         RET              ;AND RETURN
                 168 DLTC10:
009B 1C          169         INR     E        ;INC. SPACE LEFT INDICATOR
009C 2B          170         DCX     H        ;DECREMENT BUFFER POINTER
009D 4E          171         MOV     C,M      ;ECHO DELETED CHARACTER
009E CDB400  C   172         CALL    ECHO     ;
00A1 C9          173         RET              ;AND RETURN
                 174 ;
                 175 ;       DLTLIN--DELETES LINE BUFFER AND BEGINS REFILL
                 176 ;
                 177 DLTLIN:
00A2 0E23        178         MVI     C,'#'    ;ECHO A '#'
00A4 CDB400  C   179         CALL    ECHO
00A7 0E0D        180         MVI     C,CR     ;ECHO A CRLF
00A9 CDB400  C   181         CALL    ECHO
00AC 2A8000  D   182         LHLD    BUFPTR   ;GET ORIGINAL POINTER BACK
00AF D1          183         POP     D        ;GET COUNTERS BACK
00B0 D5          184         PUSH    D        ;RESAVE
00B1 C33900  C   185         JMP     GETCHR   ;GET NEW CHARACTERS
                 186 ;
                 187 ;       ECHO--ECHOES CHARACTERS TO THE TERMINAL
                 188 ;
00B4 41          189 ECHO:   MOV     B,C      ;SAVE ARGUMENT
00B5 3E1B        190         MVI     A,ESC    ;SEE IF ECHOING AN
00B7 B8          191         CMP     B        ;ESCAPE CHARACTER
00B8 C2BD00  C   192         JNZ     ECH05    ;NO--BRANCH
00BB 0E24        193         MVI     C,'$'    ;YES,ECHO AS '$'
                 194 ECH05:
00BD CDEE00  C   195         CALL    CO       ;OUTPUT IT
00C0 3E0D        196         MVI     A,CR
00C2 B8          197         CMP     B        ;CHARACTER ECHOED A CR?
00C3 C2CB00  C   198         JNZ     ECH10    ;NO--SPECIAL ACTION NOT NEEDED
00C6 0E0A        199         MVI     C,LF     ;YES--ECHO FREE LINE FEED
00C8 CDEE00  C   200         CALL    CO
                 201 ECH10:
00CB 48          202         MOV     C,B      ;RESTORE ARGUMENT
00CC C9          203         RET
                 204 ;
                 205 ;       CHKIO--CHECKS FOR CNTL-S OPERATION
                 206 ;
00CD DBED        207 CHKIO:  IN      CSTS     ;GET STATUS
00CF E602        208         ANI     RXRDY    ;CHARACTER AVAILABLE?
00D1 C8          209         RZ               ;NO,RETURN
00D2 DBEC        210         IN      CDATA    ;YES,GET CHARACTER
00D4 E67F        211         ANI     7FH      ;STRIP OFF PARITY
00D6 FE13        212         CPI     CNTLS    ;CONTROL-S?
00D8 C0          213         RNZ              ;NO,IGNORE IT
00D9 CDE200  C   214 WAIT4Q: CALL    CI       ;YES,WAIT FOR A CONTROL-Q
00DC FE11        215         CPI     CNTLQ    ;
00DE C2D900  C   216         JNZ     WAIT4Q   ;
00E1 C9          217         RET              ;GOT IT,RETURN



LOC   OBJ        SEQ         SOURCE STATEMENT

                 218 ;
                 219 ;       CI--ENTER CHARACTER FROM TERMINAL
                 220 ;
00E2 DBED        221 CI:     IN      CSTS     ;GET STATUS BYTE
00E4 E602        222         ANI     RXRDY    ;CHARACTER AVAILABLE
00E6 CAE200  C   223         JZ      CI       ;NO,LOOP
00E9 DBEC        224         IN      CDATA    ;READY,GET CHARACTER
00EB E67F        225         ANI     07FH     ;STRIP OFF PARITY
00ED C9          226         RET
                 227 ;
                 228 ;       CO--OUTPUT CHARACTER IN C REGISTER TO TERMINAL
                 229 ;
00EE DBED        230 CO:     IN      CSTS     ;GET STATUS BYTE
00F0 E601        231         ANI     TXRDY    ;TRANSMITTER READY?
00F2 CAEE00  C   232         JZ      CO       ;NO,LOOP
00F5 79          233         MOV     A,C      ;YES,MOVE CHARACTER TO ACC.
00F6 D3EC        234         OUT     CDATA    ;SEND TO TERMINAL
00F8 C9          235         RET
                 236 ;
                 237 ;       PRTBUF--PRINTS CURRENT BUFFER(CONTROL-R)
                 238 ;
                 239 PRTBUF:
00F9 0E0D        240         MVI     C,CR     ;ECHO CRLF
00FB CDB400  C   241         CALL    ECHO     ;
00FE E5          242         PUSH    H        ;SAVE CURRENT BUFFER POINTER
```

```
00FF 2A8000   D   243           LHLD    BUFPTR  ;GET POINTER TO BEGINNING
0102 D5           244           PUSH    D       ;SAVE CURRENT COUNTERS
                  245 PRLOOP:
0103 15           246           DCR     D       ;DECREMENT COUNTER
0104 FA0F01   C   247           JM      PREXIT  ;NO MORE CHARACTERS IN BUFFER
0107 4E           248           MOV     C,M     ;GET CHARACTER
0108 CDB400   C   249           CALL    ECHO    ;ECHO IT
010B 23           250           INX     H       ;INCREMENT POINTER
010C C30301   C   251           JMP     PRLOOP  ;LOOP UNTIL ALL CHARS OUTPUT
                  252 PREXIT:
010F D1           253           POP     D       ;RESTORE COUNTERS
0110 E1           254           POP     H       ;RESTORE POINTER
0111 C33900   C   255           JMP     GETCHR  ;GET NEW CHARACTER
                  256           END
```

FORTRAN COMPILER   10/12/78   PAGE   1

ISIS-II FORTRAN-80   COMPILATION OF PROGRAM UNIT MAINLN
OBJECT MODULE PLACED IN :F1:MAINLN.OBJ
COMPILER INVOKED BY: FORT80 :F1:MAINLN.FRT DEBUG DATE(10/12/78) PAGEWIDTH(78)


```
  1    (D)    SUBROUTINE MAINLN
  2           IMPLICIT LOGICAL (A-Z)
       C
       C-- MAINLINE ROUTINE FOR TEST STAND SOFTWARE. COMMAND LINE IS
       C-- SEARCHED FOR KEYLETTER AND APPROPRIATE ROUTINE IS CALLED.
       C-- ALL UNUSED LETTERS TRAP TO ERROR ROUTINE.        /
       C
       C
  3           CHARACTER LINBUF(80)*1,IMAGE*80
  4           INTEGER INDEX*2,CARRET*1,KEYLTR*1,ERRFLG*2,DUMMY*2
  5           COMMON /LINE/ LINBUF,INDEX,CARRET
  6           EQUIVALENCE (LINBUF,IMAGE)
  7           DATA CARRET /13/
       C
       C-- INITIALIZE SYSTEM
       C
  8    (E)    DUMMY=0
  9           CALL FQFSET(DUMMY,DUMMY)
       C
       C-- WRITE BANNER
       C
  10   1      WRITE(IMAGE,10,IOSTAT=ERRFLG,ERR=999) CARRET
  11   10     FORMAT('TEST STAND V0.0',A)
    (F)C
       C-- OUTPUT BUFFER
       C
  12          CALL BUFOUT(IMAGE)
       C
       C-- INITIALIZE INDEX POINTER TO START OF LINBUF
       C
  13   (G) 20 INDEX=1
       C
       C-- PROMPT OPERATOR
       C
  14          WRITE(IMAGE,30,IOSTAT=ERRFLG,ERR=999) CARRET
  15   30     FORMAT('COMMAND?',A)
  16          CALL BUFOUT(IMAGE)
       C
       C-- GET COMMAND LINE
       C
  17   (H)    CALL BUFIN(IMAGE)
       C
       C-- POSITION INDEX TO FIRST NON-BLANK CHARACTER
       C
  18   (I)    CALL DBLANK
       C
       C-- CONVERT KEYLETTER TO NORMALIZED INTEGER VALUE IE. 'A'=1
       C
  19   (J)    KEYLTR=ICHAR(LINBUF(INDEX))-#40H
  20          INDEX=INDEX+1
       C




       C-- CHECK FOR INVALID CHARACTERS
       C
  21          IF(KEYLTR.GE.1) THEN
  22          IF(KEYLTR.LE.#1AH) THEN
       C
       C-- IF VALID CHARACTER JUMP TO PROPER HANDLING ROUTINE
       C
       C        A    B    C    D    E    F    G    H    I    J    K    L    M    N
  23   (K)C  GOTO(300,100,100,300,100,100,100,100,100,100,100,100,300,100,
              C    O    P    Q    R    S    T    U    V    W    X    Y    Z
           C100,100,100,100,300,200,100,100,100,100,100,100) KEYLTR
```

2-55

```
24          ENDIF
25          ENDIF
      C
      C-- IF INVALID OUTPUT ERROR AND GET NEW LINE
      C
26          WRITE(IMAGE,40,IOSTAT=ERRFLG,ERR=999) CARRET
27    40    FORMAT('INVALID KEYLTR',A)
28          CALL BUFOUT(IMAGE)
29          GOTO 20
      C
      C-- CONTROL BRANCHES TO ONE OF THESE BASED ON KEYLETTER
      C
      C
      C-- STATEMENT LINE 100 IS USED TO TRAP ALL KEYLETTERS NOT USED
      C
30   100    WRITE(IMAGE,110,IOSTAT=ERRFLG,ERR=999) CARRET
31   110    FORMAT('NO SUCH TEST',A)
32          CALL BUFOUT(IMAGE)
33          GOTO 20
      C
      C-- TRANSITION TEST
      C
34   200    CALL TRANST
35          GOTO 20
      C
      C-- CALCULATOR MODE
      C
36   300    CALL MATH
37          GOTO 20
      C
      C-- ERROR HANDLER
      C
38   999    CALL ERROR(ERRFLG)
39          GOTO 1
40          END
```

ISIS-II FORTRAN-80   COMPILATION OF PROGRAM UNIT DBLANK
OBJECT MODULE PLACED IN :F1:DBLANK.OBJ
COMPILER INVOKED BY: FORT80 :F1:DBLANK.FRT DEBUG DATE(10/12/78) PAGEWIDTH(78)

```
1          SUBROUTINE DBLANK
2          IMPLICIT LOGICAL (A-Z)
      C
      C-- POSITIONS INDEX TO NEXT NON-BLANK CHARACTER IN LINBUF
      C
3          INTEGER INDEX*2,CARRET*1,ERRFLG*2
4          CHARACTER LINBUF(80)*1,IMAGE*80,ENDLIN*1
5          EQUIVALENCE (LINBUF,IMAGE),(ENDLIN,CARRET)
6          COMMON /LINE/ LINBUF,INDEX,CARRET
      C
7     1    IF(LINBUF(INDEX).EQ.ENDLIN) GOTO 2
8          IF(LINBUF(INDEX).NE.' ') RETURN
9          INDEX=INDEX+1
10         IF(INDEX.LE.72) GOTO 1
      C
      C-- IF END OF LINE ASK FOR MORE PARAMETERS
      C
11    2    WRITE(IMAGE,3,IOSTAT=ERRFLG,ERR=999)  CARRET
12    3    FORMAT('MISSING PARAMETER,PLEASE ENTER',A)
13         CALL BUFOUT(IMAGE)
14         CALL BUFIN(IMAGE)
15         INDEX=1
16         GOTO 1
      C
      C-- ERROR HANDLER
      C
17   999   CALL ERROR(ERRFLG)
18         RETURN
19         END
```

ISIS-II FORTRAN-80   COMPILATION OF PROGRAM UNIT ERROR
OBJECT MODULE PLACED IN :F1:ERROR.OBJ
COMPILER INVOKED BY: FORT80 :F1:ERROR.FRT DEBUG DATE(10/12/78) PAGEWIDTH(78)

```
1          SUBROUTINE ERROR(ERRNUM)
2          IMPLICIT LOGICAL(A-Z)
      C
      C-- OUTPUT ERROR MESSAGE
      C
```

```
   3              CHARACTER IMAGE*80,LINBUF(80)*1
   4              INTEGER ERRNUM*2,INDEX*2,CARRET*1
   5              EQUIVALENCE (LINBUF,IMAGE)
   6              COMMON /LINE/ LINBUF,INDEX,CARRET
       C
   7              WRITE(IMAGE,10) ERRNUM,CARRET
   8       10     FORMAT('***ERROR*** #',I4,A)
   9              CALL BUFOUT(IMAGE)
  10              RETURN
  11              END
```

```
   1    (O)        SUBROUTINE MATH
   2              IMPLICIT LOGICAL (A-Z)
       C
       C-- IMPLEMENTS CALCULATOR MODE
       C
   3              INTEGER INDEX*2,CARRET*1,ERRFLG*2
   4              CHARACTER LINBUF(80)*1,IMAGE*80,COMMND*1,SYMBOL*1
   5              REAL OP1,OP2,RESULT
   6              EQUIVALENCE (LINBUF,IMAGE)
   7              COMMON /LINE/ LINBUF,INDEX,CARRET
       C
       C-- RESCAN KEYLETTER TO DETERMINE OPERATION
       C
   8              INDEX=INDEX-1
   9    (P)        COMMND=LINBUF(INDEX)
  10              INDEX=INDEX+1
       C
       C-- MOVE INDEX TO FIRST OPERAND
       C
  11              CALL DBLANK
       C
       C-- GET IT IN
       C
  12              CALL CONVRT(OP1)
    (Q)  C
       C-- REPEAT FOR SECOND OPERAND
       C
  13              CALL DBLANK
  14              CALL CONVRT(OP2)
       C
       C-- PERFORM OPERATION
       C
  15              IF(COMMND.EQ.'M') THEN
  16              RESULT=OP1*OP2
  17              SYMBOL='*'
  18              GOTO 11
  19              ENDIF
       C
  20              IF(COMMND.EQ.'D') THEN
  21              RESULT=OP1/OP2
  22              SYMBOL='/'
  23              GOTO 11
  24              ENDIF
       C
  25              IF(COMMND.EQ.'A') THEN
  26              RESULT=OP1+OP2
  27              SYMBOL='+'
  28              GOTO 11
  29              ENDIF
       C
  30              IF(COMMND.EQ.'S') THEN




  31              RESULT=OP1-OP2
  32              SYMBOL='-'
  33              GOTO 11
  34              ENDIF
       C
       C-- OUTPUT RESULTS
       C
  35       11     WRITE(IMAGE,12,IOSTAT=ERRFLG,ERR=999) OP1,SYMBOL,OP2,RESULT,
              1CARRET
  36       12     FORMAT(F18.5,1X,A,1X,F18.5,1X,'=',1X,F18.5,A)
  37              CALL BUFOUT(IMAGE)
```

```
38          RETURN
    C
    C-- ERROR HANDLER
    C
39      999 CALL ERROR(ERRFLG)
40          RETURN
41          END
```

```
1     (R)   SUBROUTINE CONVRT(VALUE)
2           IMPLICIT LOGICAL(A-Z)
    C
    C-- INPUTS NEXT PARAMETER IN LINBUF
    C
3           INTEGER I*1,INDEX*2,TMPIND*1,CARRET*1,ERRFLG*2
4           REAL VALUE
5           CHARACTER LINBUF(80)*1,TMPBUF(20)*1,BUFFER*20,ENDLIN*1
6           EQUIVALENCE (TMPBUF,BUFFER),(ENDLIN,CARRET)
7           COMMON /LINE/ LINBUF,INDEX,CARRET
    C
    C-- INITIALIZE
    C
8           DO 21 I=1,19
9      21   TMPBUF(I)=' '
10          TMPBUF(20)=ENDLIN
11          TMPIND=1
    C
    C-- FILL BUFFER UNTIL COMMA OR ENDLINE ENCOUNTERED
    C
12     22   TMPBUF(TMPIND)=LINBUF(INDEX)
13          INDEX=INDEX+1
14          TMPIND=TMPIND+1
15          IF(LINBUF(INDEX).EQ.',') THEN
16    (S)   INDEX=INDEX+1
17          GOTO 23
18          ENDIF
19          IF(LINBUF(INDEX).EQ.ENDLIN) GOTO 23
20          GOTO 22
    C
    C-- READ UNDER FORMAT CONTROL
    C
21    (T) 23 READ(BUFFER,24,IOSTAT=ERRFLG,ERR=999) VALUE
22      24  FORMAT(F19.5)
23          RETURN
    C
    C-- ERROR HANDLER
    C
24      999 CALL ERROR(ERRFLG)
25          RETURN
26          END
```

```
1           SUBROUTINE TRANST
2           IMPLICIT LOGICAL (A-Z)
    C
    C-- PERFORM TRANSITION TESTING
    C
3           REAL START,STOP,STEP,TOL,TEMP,VOLTAG,VCC(20),
           1LOWLVL(20),LOTOHI(20),HILVL(20),HITOLO(20)
    C
4           INTEGER CARRET*1,ITEMP*2,TSTINP*2,SAMPLE*2,
           1LSTSAM*2,DELTA*2,ERRFLG*2,PNTCNT*1,INDEX*2,I*1
    C
5           CHARACTER LINBUF(80)*1,IMAGE*80,TEST(20)*6
    C
6           EQUIVALENCE (LINBUF,IMAGE)
    C
7           COMMON /LINE/ LINBUF,INDEX,CARRET
    C
    C-- INITIALIZE
    C
```

```
      8              DO 5, I=1,20
      9       5      TEST(I)='PASSED'
     10              TSTINP=0
     11              PNTCNT=1
              C
              C-- SCAN COMMAND TAIL FOR PARAMETERS
              C
              C      VCC  START      STOP      STEP      TOLERANCE
              C
     12              CALL DBLANK
     13              CALL CONVRT(START)
     14              CALL DBLANK
     15              CALL CONVRT(STOP)
     16              CALL DBLANK
     17              CALL CONVRT(STEP)
     18              CALL DBLANK
     19              CALL CONVRT(TOL)
              C
              C-- IF (STOP-START)/STEP YIELDS MORE THAN 20 STEPS
              C-- OUTPUT MESSAGE AND RETURN
              C
     20              IF(IFIX((STOP-START)/STEP).GT.20) THEN
     21              WRITE(IMAGE,10,IOSTAT=ERRFLG,ERR=999) CARRET
     22      10      FORMAT('TOO MANY POINTS',A)
     23              CALL BUFOUT(IMAGE)
     24              RETURN
     25              ENDIF
              C
              C-- GET TEMPERATURE AND LINEARIZE
              C
     26              CALL ADCIN(ITEMP,0)
     27     (U)      TEMP=98.63*ALOG(FLOAT(ITEMP))+13.56




              C
              C-- OUTPUT HEADER
              C
     28              WRITE(IMAGE,20,IOSTAT=ERRFLG,ERR=999) TOL,TEMP,CARRET,CARRET
     29      20      FORMAT('TRANSITION TEST   TOLERANCE=',F5.1,
              1'% AMBIENT TEMPERATURE = ',F6.2,' DEGREES C',A,A)
     30              CALL BUFOUT(IMAGE)
     31              WRITE(IMAGE,30,IOSTAT=ERRFLG,ERR=999) CARRET,CARRET
     32      30      FORMAT('   VCC    HIGH TRANS  LOW TRANS  HIGH    LOW     TEST',
              1A,A)
     33              CALL BUFOUT(IMAGE)
              C
              C-- BEGIN TEST; OUTPUT STARTING VCC VALUE
              C
     34              VOLTAC=START
     35      40      VCC(PNTCNT)=VOLTAG
     36              CALL DACOUT(IFIX(VOLTAG*409.6),0)
              C
              C-- OUTPUT ZERO VOLTS TO TEST INPUT
              C
     37     (V)      CALL DACOUT(TSTINP,1)
              C
              C-- GET ONE SAMPLE
              C
     38              CALL ADCIN(SAMPLE,1)
              C
              C-- MAKE IT THE LAST SAMPLE AND ALSO STORE IT
              C
     39              LSTSAM=SAMPLE
     40              LOWLVL(PNTCNT)=FLOAT(SAMPLE)*409.6
              C
              C-- BEGIN LOOP LOOKING FOR LOW TO HIGH TRANSITION
              C
     41      50      TSTINP=TSTINP+1
     42              CALL DACOUT(TSTINP,1)
              C
              C-- GET SAMPLE
              C
     43              CALL ADCIN(SAMPLE,1)
     44              DELTA=SAMPLE-LSTSAM
              C
              C-- SEE IF TRANSITION;DELTA .GT. 2.2 VOLTS
              C
     45              IF(DELTA.LT.901) THEN
     46              LSTSAM=SAMPLE
              C
              C-- NO TRANSITION; IF TSTINP NOW UP TO 5.5V AND NO TRANSITION
              C-- OUTPUT MESSAGE INDICATING DEAD PART AND RETURN
              C
     47              IF(TSTINP.GE.2251) THEN
```

```
48                    WRITE(IMAGE,60,IOSTAT=ERRFLG,ERR=999) CARRET
49   (W)  60          FORMAT('DEAD PART, NO TRANSITION',A)
50                    CALL BUFOUT(IMAGE)
51                    RETURN
52                    ENDIF
         C
         C-- CONTINUE LOOP




         C
53               GOTO 50
54               ENDIF
         C
         C-- TRANSITION; ASSIGN ARRAY ELEMENT
         C
55               LOTOHI(PNTCNT)=FLOAT(TSTINP)/409.6
         C
         C-- CHECK TOLERANCE
         C
56               IF((LOTOHI(PNTCNT).GE.(.8-(TOL/100.*.8))).AND.
         1(LOTOHI(PNTCNT).LE.(.8+(TOL/100.*.8)))) GOTO 70
         C
         C-- TEST FAILED
         C
57               TEST(PNTCNT)='FAILED'
         C
         C-- BEGIN HIGH TO LOW TEST
         C
         C
         C-- OUTPUT 5.0 VOLTS
         C
58   70          TSTINP=2048
59               CALL DACOUT(TSTINP,1)
         C
         C-- GET SAMPLE
         C
60               CALL ADCIN(SAMPLE)
         C
         C-- MAKE IT LAST SAMPLE AND ALSO STORE IT
         C
61               LSTSAM=SAMPLE
62               HILVL(PNTCNT)=FLOAT(SAMPLE)*409.6
         C
         C-- BEGIN LOOP LOOKING FOR HIGH TO LOW TRANSITION
         C
63   80          TSTINP=TSTINP-1
64               CALL DACOUT(TSTINP,1)
         C
         C-- GET SAMPLE
         C
65               CALL ADCIN(SAMPLE,1)
66               DELTA=LSTSAM-SAMPLE
         C
         C-- SEE IF TRANSITION; DELTA .GT. 2.2 VOLTS
         C
67               IF(DELTA.LT.901) THEN
68               LSTSAM=SAMPLE
         C
         C-- NO TRANSITION; CHECK TO SEE IF VOLTAGE DOWN TO ZERO
         C
69               IF(TSTINP.LE.0) THEN
         C
         C-- YES; OUTPUT DEAD PART MESSAGE
         C
70               WRITE(IMAGE,60,IOSTAT=ERRFLG,ERR=999) CARRET
71               CALL BUFOUT(IMAGE)

72               RETURN
73               ENDIF
         C
         C-- CONTINUE LOOP
         C
74               GOTO 80
75               ENDIF
         C
         C-- TRANSITION; ASSIGN ARRAY ELEMENT
         C
76               HITOLO(PNTCNT)=FLOAT(TSTINP)*409.6
         C
         C-- CHECK TOLERANCE
         C
77               IF((HITOLO(PNTCNT).GE.(3.5-(TOL/100.*3.5))).AND.
         1(HITOLO(PNTCNT).LE.(3.5+(TOL/100.*3.5)))) GOTO 90
         C
         C-- TEST FAILED
         C
```

```
 78          TEST(PNTCNT)='FAILED'
       C
       C-- INCREMENT VCC AND IF NOT .GT. STOP CONTINUE
       C
 79      90   VOLTAG=VOLTAG+STEP
 90          IF(VOLTAG.LE.STOP) THEN
 81          PNTCNT=PNTCNT+1
 82          TSTINP=0
 83          GOTO 40
 84          ENDIF
       C
       C-- TEST COMPLETE; OUTPUT RESULTS
       C
 85          DO 110,I=1,PNTCNT
 86          WRITE(IMAGE,100,IOSTAT=ERRFLG,ERR=999) VCC(I),
            1LOTOHI(I),HITOLO(I),HILVL(I),LOWLVL(I),TEST(I),CARRET
 87     100   FORMAT(3X,F5.2,3X,F6.2,6X,F6.2,3X,F6.2,1X,F6.2,2X,6A,A)
 88     110   CALL BUFOUT(IMAGE)
 89          RETURN
       C
       C-- ERROR HANDLER
       C
 90     999   CALL ERROR(ERRFLG)
 91          RETURN
 92          END
```

```
  1   (X)       SUBROUTINE ADCIN(VALUE,CHAN)
       C
       C-- ROUTINE TO INPUT SINGLE VALUE FROM A/D CONVERTER CHANNEL
       C-- GIVEN AND RETURN IT IN VALUE FIELD.
       C
  2          INTEGER*2 VALUE
  3          INTEGER*1 CHAN
  4   $INCLUDE(:F1:ADCDAC.DEC)
    = C
    = C-- DEFINITIONS OF ISBC 732 REGISTERS
    = C
    = C
    = C-- COMMAND STATUS REGISTER
    = C
  5 =        INTEGER*1 CMDSTS
    = C
    = C-- MUX ADDRESS REGISTER
    = C
  6 =        INTEGER*1 MUXADR
    = C
    = C-- LAST CHANNEL REGISTER
    = C
  7 =        INTEGER*1 LSTCHN
    = C
    = C-- CLEAR INTERRUPT COMMAND WORD
    = C
  8 =        INTEGER*1 CLRINT
    = C
    = C-- ADC DATA REGISTER
    = C
  9 =        INTEGER*2 ADCDAT
    = C
    = C-- DAC 0 DATA REGISTER
    = C
 10 =        INTEGER*2 DAC0
    = C
    = C-- DAC 1 DATA REGISTER
    = C
 11 =        INTEGER*2 DAC1
    = C
    = C-- SET UP COMMON BLOCK
    = C
 12 =        COMMON /ADC/ CMDSTS,MUXADR,LSTCHN,CLRINT,ADCDAT,DAC0,DAC1
       C
       C-- SET UP CHANNEL ADDRESS
       C
 13          MUXADR=CHAN
       C
       C-- START CONVERSION
       C
 14          CMDSTS=#1H
```

```
        C
        C-- WAIT FOR END OF CONVERSION
        C
15        1   IF((CMDSTS.AND.#80H).NE.#80H) GOTO 1
        C
        C-- GET DATA IN
        C
16            VALUE=ADCDAT
        C
        C-- RIGHT JUSTIFY AND CONVERT TO COUNTS
        C
17            VALUE=VALUE/16
18            IF(VALUE.LT.0) VALUE=VALUE+4096+1
19            RETURN
20            END
```

```
1   (Y)       SUBROUTINE DACOUT(VALUE,CHAN)
        C
        C-- OUTPUTS VALUE TO D/A CONVERTER
        C
2             INTEGER*2 VALUE,CHAN
3       $INCLUDE(:F1:ADCDAC.DEC)
    = C
    = C-- DEFINITIONS OF ISBC 732 REGISTERS
    = C
    = C
    = C-- COMMAND STATUS REGISTER
    = C
4   =         INTEGER*1 CMDSTS
    = C
    = C-- MUX ADDRESS REGISTER
    = C
5   =         INTEGER*1 MUXADR
    = C
    = C-- LAST CHANNEL REGISTER
    = C
6   =         INTEGER*1 LSTCHN
    = C
    = C-- CLEAR INTERRUPT COMMAND WORD
    = C
7   =         INTEGER*1 CLRINT
    = C
    = C-- ADC DATA REGISTER
    = C
8   =         INTEGER*2 ADCDAT
    = C
    = C-- DAC 0 DATA REGISTER
    = C
9   =         INTEGER*2 DAC0
    = C
    = C-- DAC 1 DATA REGISTER
    = C
10  =         INTEGER*2 DAC1
    = C
    = C-- SET UP COMMON BLOCK
    = C
11  =         COMMON /ADC/ CMDSTS,MUXADR,LSTCHN,CLRINT,ADCDAT,DAC0,DAC1
        C
        C-- OUTPUT VALUE TO PROPER CHANNEL
        C-- AFTER SHIFTING INTO HIGH ORDER 12 BITS
        C
12            IF(VALUE.LT.0) VALUE=VALUE+4096+1
13            VALUE=VALUE*16
14            IF(CHAN.EQ.0) DAC0=VALUE
15            IF(CHAN.EQ.1) DAC1=VALUE
16            RETURN
17            END
```

APPENDIX B
CODE LISTINGS

ISIS-II PL/M-80 V3.1 COMPILATION OF MODULE SEMAPHORES
OBJECT MODULE PLACED IN :F1:SEMMOD.OBJ
COMPILER INVOKED BY: plm80 :F1:SEMMOD.plm DEBUG DATE(10/12/78) PAGEWIDTH(78)


```
    1           SEMAPHORES:
                    DO;

                /**********************************************************

                    Contains LOCK and UNLOCK procedures for
                    manipulating semaphores. Called by FORTRAN
                    routines with one parameter; the address of
                    the index of the semaphore to be operated on.

                **********************************************************/

                $nolist

   17    1       DECLARE (stslok,setlok,dsklok) (10) BYTE PUBLIC;
   18    1       DECLARE semaphore(3) ADDRESS PUBLIC DATA(
                    .stslok,
                    .setlok,
                    .dsklok);
   19    1       DECLARE token (3) STRUCTURE(
                    link ADDRESS,
                    length ADDRESS,
                    type ADDRESS) PUBLIC;

   20    1       LOCK: PROCEDURE(sema$number$ptr) REENTRANT PUBLIC;

   21    2       DECLARE sema$number$ptr ADDRESS;
   22    2       DECLARE sema$number BASED sema$number$ptr BYTE;
   23    2       DECLARE msg$ptr ADDRESS;

   24    2          msg$ptr=RQWAIT(semaphore(sema$number),0);
   25    2 (A)      RETURN;
   26    2          END;

   27    1       UNLOCK: PROCEDURE(sema$number$ptr) REENTRANT PUBLIC;

   28    2       DECLARE sema$number$ptr ADDRESS;
   29    2       DECLARE sema$number BASED sema$number$ptr BYTE;

   30    2          CALL RQSEND(semaphore(sema$number),.token(sema$number));
   31    2 (B)      RETURN;
   32    2          END;
   33    1     END SEMAPHORES;
```


ISIS-II FORTRAN-80   COMPILATION OF PROGRAM UNIT STSINP
OBJECT MODULE PLACED IN :F1:STSMOD.OBJ
COMPILER INVOKED BY: FORT80 :F1:STSMOD.FRT DEBUG DATE(10/12/78) PAGEWIDTH(78)


```
    1  (C)    SUBROUTINE STSINP
    2         IMPLICIT LOGICAL (A-Z)
           C
           C-- TASK CODE FOR STATUS INPUT TASK. UPDATES STATUS COMMON
           C-- BLOCK WITH ANALOG AND DIGITAL DATA VALUES. ALSO DOES
           C-- ANALOG COUNT TO ENGINEERING UNIT CONVERSIONS.
           C
    3         CHARACTER SMPLBF*22,CLOCK*12
    4         INTEGER*2 SAMPLS(11),DUMMY
    5         REAL ANDATA(11)
    6         EQUIVALENCE (SMPLBF,SAMPLS)
    7         INTEGER*1 DIGDAT,I
    8         COMMON /STATUS/ ANDATA,DIGDAT,CLOCK
           C
           C-- INITIALIZE FLOATING POINT LIBRARIES
           C
    9  (D)    DUMMY=0
   10         CALL FQFSET(DUMMY,DUMMY)
           C
           C-- CALL INITIALIZATION ROUTINE
           C
   11  (E)    CALL INITIO
           C
           C-- CALL ROUTINE TO INPUT SAMPLES
           C
   12  10     CALL SMPLIN(SMPLBF)
       (F) C
           C-- SHIFT SAMPLES TO RIGHT JUSTIFY
           C
```

```
13          DO 50 I=1,11
14 (G)       SAMPLS(I)=SAMPLS(I)/16
15    50    IF(SAMPLS(I).LT.0) SAMPLS(I)=SAMPLS(I)+4096+1
      C
      C-- WAIT FOR ACCESS TO STATUS COMMON BLOCK FOR UPDATE
      C-- THEN CONVERT SAMPLES TO ENGINEERING UNITS AND STORE
      C
16         ⌈CALL LOCK(0)
17          ANDATA(1)=FLOAT(SAMPLS(1))
18          ANDATA(2)=ALOG10(FLOAT(SAMPLS)*2.34)-365.98
19          ANDATA(3)=ALOG10(FLOAT(SAMPLS(3)/13.9)-21.53
20          ANDATA(4)=13.23*FLOAT(SAMPLS(4))-20.78
21          ANDATA(5)=FLOAT(SAMPLS(5))
22 (H)      ANDATA(6)=FLOAT(SAMPLS(6))/14.225
23          ANDATA(7)=FLOAT(SAMPLS(7))
24          ANDATA(8)=ALOG(FLOAT(SAMPLS(8)/23.98)+235.98
25          ANDATA(9)=FLOAT(SAMPLS(9))
26          ANDATA(10)=FLOAT(SAMPLS(10))
27          ANDATA(11)=(FLOAT(SAMPLS(11))-119.34)/5.734
28         ⌊CALL INPUT(#0E8H,DIGDAT)
      C
      C-- RELEASE LOCK ON STATUS COMMON BLOCK




      C
29          CALL UNLOCK(0)
      C
      C-- DELAY FOR 1 SECOND THEN SCAN AGAIN
      C
30 (I)       CALL WAIT
      C
      C-- LOOP BACK
      C
31          GOTO 10
32          END
```

PL/M-80 COMPILER   10/12/78   PAGE   1

ISIS-II PL/M-80 V3.1 COMPILATION OF MODULE ANALOGIOMOD
OBJECT MODULE PLACED IN :F1:aiomod.OBJ
COMPILER INVOKED BY: plm80 :F1:aiomod.plm DEBUG DATE(10/12/78) PAGEWIDTH(78)


```
 1          ANALOG$IO$MOD:
                DO;

            /**********************************************************

               Inputs analog samples into buffer provided as
               calling parameter.

            **********************************************************/

            $nolist
24    1 (J) DECLARE AN$RESP (10) BYTE PUBLIC;
25    1     DECLARE ANALOG$REQUEST$MESSAGE ai$msg;

26    1     INITIO: PROCEDURE PUBLIC;

            /* initializes mesage to be used for analog samples */

27    2    ⌈ANALOG$REQUEST$MESSAGE.length=size(ANALOG$REQUEST$MESSAGE);
28    2    │ANALOG$REQUEST$MESSAGE.type=AIS$QS;
29    2 (K)│ANALOG$REQUEST$MESSAGE.response$exchange=.AN$RESP;
30    2    │ANALOG$REQUEST$MESSAGE.base$ptr=0FFF0H;
31    2    ⌊ANALOG$REQUEST$MESSAGE.channel$gain=0;

32    2     RETURN;
33    2     END; /* of INIT$IO */

34    1     SMPL$IN: PROCEDURE(sample$buffer$ptr,buf$size) PUBLIC;

            /* inputs buf$size/2 analog word samples */

35    2     DECLARE (sample$buffer$ptr,buf$size,dummy) ADDRESS;
36    2     DECLARE sample$buffer BASED sample$buffer$ptr (1) BYTE;

37    2         ANALOG$REQUEST$MESSAGE.array$ptr=sample$buffer$ptr;
38    2         ANALOG$REQUEST$MESSAGE.count=buf$size/2;

39    2 (L)     CALL RQSEND(.RQAIEX,.ANALOG$REQUEST$MESSAGE);
40    2         dummy=RQWAIT(.AN$RESP,0);
```

```
41    2           RETURN;
42    2           END; /* of SMPLSIN */

43    1       END ANALOG$IO$MOD;
```

```
 1           SUBROUTINE SCAN
         C
         C-- CODE FOR SCAN TASK THAT COMPARES STATUS VALUES WITH
         C-- SETPOINTS AND SETS OPERATOR ALARMS ACCORDINGLY. ALSO
         C-- LOGS DISK RECORD OF STATUS WHEN MIN5UP FLAG IS TRUE.
         C
 2       $INCLUDE(:F1:EQUIV.DEC)
 3    =       CHARACTER BUFFER*57,PARAMS(57)*1
 4    =       REAL PH,VOLUME,TEMP,DISOXY,TOTCAR,ORGCAR
 5    =       REAL SUSSOL,PHOSFT,INFLOW,EFLFLO,TURBID
 6    =       INTEGER*1 DIGDAT
 7    =       INTEGER*2 MONTH,DAY,YEAR,HOUR,MINUTE,SECOND
 8    =       EQUIVALENCE (PARAMS,BUFFER)
 9    =       EQUIVALENCE (PARAMS,PH)
10    =       EQUIVALENCE (PARAMS(5),VOLUME)
11    =       EQUIVALENCE (PARAMS(9),TEMP)
12    =       EQUIVALENCE (PARAMS(13),DISOXY)
13    =       EQUIVALENCE (PARAMS(17),TOTCAR)
14    =       EQUIVALENCE (PARAMS(21),ORGCAR)
15    =       EQUIVALENCE (PARAMS(25),SUSSOL)
16    =       EQUIVALENCE (PARAMS(29),PHOSFT)
17    =  (M)  EQUIVALENCE (PARAMS(33),INFLOW)
18    =       EQUIVALENCE (PARAMS(37),EFLFLO)
19    =       EQUIVALENCE (PARAMS(41),TURBID)
20    =       EQUIVALENCE (PARAMS(45),DIGDAT)
21    =       EQUIVALENCE (PARAMS(46),MONTH)
22    =       EQUIVALENCE (PARAMS(48),DAY)
23    =       EQUIVALENCE (PARAMS(50),YEAR)
24    =       EQUIVALENCE (PARAMS(52),HOUR)
25    =       EQUIVALENCE (PARAMS(54),MINUTE)
26    =       EQUIVALENCE (PARAMS(56),SECOND)
27           INTEGER*2 ERRFLG,RECNO,DUMMY
28           REAL SETSOL,SETCAR,SETPHS,SETTRB
29           INTEGER*1 MIN5UP
30           COMMON /MIN5/ MIN5UP
31           COMMON /SETPNT/ SETPHS,SETSOL,SETCAR,SETTRB
32           COMMON /STATUS/ BUFFER
33           COMMON /LSTREC/ RECNO
         C
         C-- INITIALIZE RECORD COUNTER
         C
34           RECNO=1
         C
         C-- INITIALIZE MATH LIBRARIES
         C
35           DUMMY=0
36           CALL FQFSET(DUMMY,DUMMY)
         C
         C-- WAIT FOR ACCESS TO STATUS AND SETPOINT COMMON BLOCKS
         C
37    10     CALL LOCK(0)




38           CALL LOCK(1)
         C
         C-- SCAN FOR ALARMS ONLY IF EFFLUENT PUMP IS ON
         C
39           IF((DIGDAT.AND.#04H).EQ.#04H) THEN
40           IF(PHOSFT.GT.SETPHS) THEN
41           CALL OUTPUT(#0EBH,#01H)
42           ELSE
43           CALL OUTPUT(#0EBH,#00H)
44           ENDIF
45           IF(SUSSOL.GT.SETSOL) THEN
46    (N)    CALL OUTPUT(#0EBH,#03H)
47           ELSE
48           CALL OUTPUT(#0EBH,#02H)
49           ENDIF
50           IF(TOTCAR.GT.SETCAR) THEN
51           CALL OUTPUT(#0EBH,#05H)
52           ELSE
```

```
53          CALL OUTPUT(#0EBH,#04H)
54          ENDIF
55   (N)    IF(TURBID.GT.SETTRB) THEN
56          CALL OUTPUT(#0EBH,#07H)
57          ELSE
58          CALL OUTPUT(#0EBH,#06H)
59          ENDIF
60          ENDIF
     C
     C-- IF MIN5 TASK HAS SET MIN5UP LOG STATUS ON DISK
     C
61   (O)    IF(MIN5UP.NE.0) THEN
62          MIN5UP=0
     C
     C-- WAIT FOR ACCESS TO DISK
     C
63          CALL LOCK(2)
64          OPEN(3,FILE=':D0:TODAYS.RPT',STATUS='OLD',IOSTAT=ERRFLG,
            1ERR=9000,ACCESS='DIRECT',RECL=57)
65   (P)    WRITE(3,REC=RECNO,IOSTAT=ERRFLG,ERR=9100) BUFFER
66          RECNO=RECNO+1
67          CLOSE (3,IOSTAT=ERRFLG,ERR=9200)
68          CALL UNLOCK(2)
69          ENDIF
     C
     C-- RELEASE LOCK ON STATUS AND SETPOINT COMMON BLOCKS
     C
70          CALL UNLOCK(1)
71          CALL UNLOCK(0)
     C
     C-- DELAY FOR 1 SECOND THEN SCAN AGAIN
     C
72          CALL WAIT
     C
     C-- LOOP BACK
     C
73          GOTO 10
     C
     C-- ERROR HANDLERS



(Q)  C
74   9000 WRITE(6,9001) ERRFLG
75   9001 FORMAT('OPEN ERROR IN SCAN; #',I4)
76        GOTO 10
77   9100 WRITE(6,9101) ERRFLG
78   9101 FORMAT('WRITE ERROR IN SCAN; #',I4)
79        GOTO 10
80   9200 WRITE(6,9201) ERRFLG
81   9201 FORMAT('CLOSE ERROR IN SCAN; #',I4)
82        GOTO 10
83        END
```

PL/M-80 COMPILER   10/12/78   PAGE   1

ISIS-II PL/M-80 V3.1 COMPILATION OF MODULE MIN5MOD
OBJECT MODULE PLACED IN :F1:MIN5MD.OBJ
COMPILER INVOKED BY: plm80 :F1:MIN5MD.plm DEBUG DATE(10/12/78) PAGEWIDTH(78)


```
1        MIN$5$MOD:
            DO;

         /********************************************************

            This module contains the code for TIMER$5 who
            waits for 5 minutes and sets a flag telling
            SCAN to log a report on the disk, and for
            WAIT who waits for 1 second then returns

         ********************************************************/

         $nolist

19   1   DECLARE min$5$ex (10) BYTE PUBLIC;
20   1   DECLARE min$5$up BYTE AT(0FFEEH);
21   1   DECLARE time$out$msg$ptr ADDRESS;
22   1   DECLARE five$minute$delay$count LITERALLY '6000';
23   1   DECLARE time$sup LITERALLY '01H';

24   1   WAIT: PROCEDURE REENTRANT PUBLIC;

25   2   (R)  time$out$msg$ptr=RQWAIT(.min$5$ex,20);
26   2        RETURN;
```

```
27    2         END;

28    1         TIMER5: PROCEDURE PUBLIC;
29    2             min$5$up=0;

                /* enter task loop */

30    2             DO WHILE 1;
31    3  (S)        time$out$msg$ptr=RQWAIT(.min$5$ex,five$minute$delay$count);
32    3             min$5$up=times$up;
33    3             END; /* of do while 1 */
34    2         END; /* of procedure */
35    1         END; /* of module */
```

```
1             REPORT:
                DO;

        /*************************************************************

            This module contains the code for the REPORT
            task that prints formatted reports of system
            status upon command. Commands come in from
            PRTREQ exchange with type=100 for todays
            status report and type = 101 for yesterday's
            status report. PRINT is the FORTRAN routine
            that does the actual work.

        *************************************************************/

                $nolist

21    1         PRINT: PROCEDURE (file$ptr,name$size,request$type) EXTERNAL;
22    2             DECLARE (file$ptr,name$size) ADDRESS;
23    2             DECLARE request$type BYTE;
24    2         END PRINT;

25    1         FQFSET: PROCEDURE(A,ERRH) EXTERNAL;
26    2         DECLARE (A,ERRH) ADDRESS;
27    2         END FQFSET;

28    1         DECLARE prt$req (10) BYTE PUBLIC;

29    1         REPORT: PROCEDURE PUBLIC;

30    2         DECLARE today$type LITERALLY '100';
31    2         DECLARE yesterday$type LITERALLY '101';
32    2         DECLARE (ptr,dummy) ADDRESS;
33    2         DECLARE msg BASED ptr STRUCTURE(
                    link ADDRESS,
                    length ADDRESS,
                    type BYTE,
                    home$exchange ADDRESS,
                    response$exchange ADDRESS);
34    2         DECLARE today$file$name (*) BYTE DATA(':D0:TODAYS.RPT');
35    2         DECLARE ystday$file$name (*) BYTE DATA(':D0:YSTDAY.RPT');

                /* initialize math handler */

36    2         dummy=0;
37    2         CALL FQFSET(.dummy,.dummy);

                /* enter task loop */

38    2             DO WHILE 1;
39    3  (T)        ptr=RQWAIT(.prt$req,0);




40    3             IF msg.type=today$type THEN
41    3                 CALL print(.today$file$name,SIZE(today$file$name),.msg.t
      -    ype);
42    3             ELSE IF msg.type=yesterday$type THEN
43    3                 CALL print(.ystday$file$name,size(ystday$file$name),.msg
      -    .type);
                    CALL RQSEND(msg.response$exchange,ptr);
45    3             END; /* of do while */
46    2         END; /* of task */
47    1         END REPORT;
```

```
    1           SUBROUTINE HEADER
          C
          C-- CALLED BY PRINT TO OUTPUT REPORT HEADER
          C
    2           WRITE(6,200)
    3     200   FORMAT(' DATE      TIME    PH    VOLUME     TEMP    DISSOLVED
                -',
                1'TOTAL  ORGANIC  SUSPENDED PHOSPHATE INFLUENT EFFLUENT ',
                2'TURBID  AIR DIS MIX INF')
    4           WRITE(6,201)
    5     201   FORMAT(44X,'OXYGEN     CARBON   CARBON     SOLIDS     CONC',6X,
                1'FLOW      FLOW')
    6           WRITE(6,202)
    7     202   FORMAT(24X,'(CU.M)       (C)       (MG/ML)   (MG/ML) (MG/ML)   '
                -',
                1'(MG/ML)    (MG/ML)   (MG/ML)  (MG/ML)   %')
    8           RETURN
    9           END
```

```
          C
          C-- SUBROUTINE PRINT CALLED BY REPORT TO GENERATE FORMATTED
          C-- REPORTS.  PRINTS EITHER TODAY'S FILE OR YESTERDAY'S
          C-- DEPENDING ON FILNM INPUT VALUE.
          C
    1           SUBROUTINE PRINT(FILNM,TYPE)
    2           IMPLICIT LOGICAL (A-Z)
    3           CHARACTER*14 FILNM
    4           INTEGER*2 ERRFLG,RECCNT,LSTREC
    5           INTEGER*1 TYPE
    6           INTEGER*1 INDEX
    7     $INCLUDE(:F1:EQUIV.DEC)
    8   =       CHARACTER BUFFER*57,PARAMS(57)*1
    9   =       REAL PH,VOLUME,TEMP,DISOXY,TOTCAR,ORGCAR
   10   =       REAL SUSSOL,PHOSFT,INFLOW,EFLFLO,TURBID
   11   =       INTEGER*1 DIGDAT
   12   =       INTEGER*2 MONTH,DAY,YEAR,HOUR,MINUTE,SECOND
   13   =       EQUIVALENCE (PARAMS,BUFFER)
   14   =       EQUIVALENCE (PARAMS,PH)
   15   =       EQUIVALENCE (PARAMS(5),VOLUME)
   16   =       EQUIVALENCE (PARAMS(9),TEMP)
   17   =       EQUIVALENCE (PARAMS(13),DISOXY)
   18   =       EQUIVALENCE (PARAMS(17),TOTCAR)
   19   =       EQUIVALENCE (PARAMS(21),ORGCAR)
   20   =       EQUIVALENCE (PARAMS(25),SUSSOL)
   21   =       EQUIVALENCE (PARAMS(29),PHOSFT)
   22   =       EQUIVALENCE (PARAMS(33),INFLOW)
   23   =       EQUIVALENCE (PARAMS(37),EFLFLO)
   24   =       EQUIVALENCE (PARAMS(41),TURBID)
   25   =       EQUIVALENCE (PARAMS(45),DIGDAT)
   26   =       EQUIVALENCE (PARAMS(46),MONTH)
   27   =       EQUIVALENCE (PARAMS(48),DAY)
   28   =       EQUIVALENCE (PARAMS(50),YEAR)
   29   =       EQUIVALENCE (PARAMS(52),HOUR)
   30   =       EQUIVALENCE (PARAMS(54),MINUTE)
   31   =       EQUIVALENCE (PARAMS(56),SECOND)
   32           CHARACTER*3 AIR,MIX,INFLNT,DISCHG
   33           COMMON /LSTREC/ LSTREC
          C
          C-- INITIALIZE RECORD COUNT
          C
   34           RECCNT=1
          C
          C-- INITIALIZE INDEX
          C
   35           INDEX=1
          C
          C-- OUTPUT HEADER
          C
   36           CALL HEADER
          C
```

```
      C-- WAIT FOR FILE ACCESS IF TODAY'S FILE
      C
37    1     IF(TYPE.EQ.100) CALL LOCK(2)
38          OPEN(8,FILE=FILNM,STATUS='OLD',IOSTAT=ERRFLG,
           1ERR=9000,ACCESS='DIRECT',RECL=57)
39    10    READ(8,REC=RECCNT,IOSTAT=ERRFLG,ERR=9100) BUFFER
40          RECCNT=RECCNT+1
41          IF((DIGDAT.AND.#01H).EQ.#01H) THEN
42          AIR=' ON'
43          ELSE
44          AIR='OFF'
45          ENDIF
46          IF((DIGDAT.AND.#02H).EQ.#02H) THEN
47          MIX=' ON'
48          ELSE
49          MIX='OFF'
50          ENDIF
51          IF((DIGDAT.AND.#04H).EQ.#04H) THEN
52          DISCHG=' ON'
53          ELSE
54          DISCHG='OFF'
55          ENDIF
56          IF((DIGDAT.AND.#08H).EQ.#08H) THEN
57          INFLNT=' ON'
58          ELSE
59          INFLNT='OFF'
60          ENDIF
61          WRITE(6,101) MONTH,DAY,YEAR,HOUR,MINUTE,SECOND,
           1PH,VOLUME,TEMP,DISOXY,TOTCAR,ORGCAR,SUSSOL,PHOSFT,
           2INFLOW,EFLFLO,TURBID,AIR,DISCHG,MIX,INFLNT
62    101   FORMAT(I2,'/',I2,'/',I2,1X,I2,':',I2,':',I2,1X,F4.1,1X,F9.2,
           Z1X,F9.4,1X,F9.4,1X,F8.3,1X,F8.3,1X,F9.4,1X,F9.4,1X,F8.3,1X,F8.3
           -,1X,F7.3,
           Z1X,A3,1X,A3,1X,A3,1X,A3)
      C
      C-- CHECK FOR END OF FILE AND OTHER THINGS
      C
63          INDEX=INDEX+1
64          IF(TYPE.EQ.100) THEN
65          IF(INDEX.LE.10) THEN
66          IF(RECCNT.LT.LSTREC) THEN
67          GOTO 10
68          ELSE
69          CLOSE(8,IOSTAT=ERRFLG,ERR=9200)
70          CALL UNLOCK(2)
71          RETURN
72          ENDIF
73          ELSE
74          INDEX=1
75          CLOSE(8,IOSTAT=ERRFLG,ERR=9200)
76          CALL UNLOCK(2)
77          GOTO 1
78          ENDIF
79          ELSE
80          IF(RECCNT.LE.288) THEN
81          GOTO 10
82          ELSE
83          CLOSE(8,IOSTAT=ERRFLG,ERR=9200)
84          RETURN
85          ENDIF
86          ENDIF
      C
      C-- ERROR HANDLERS
      C
87    9000  WRITE(6,9001) ERRFLG
88    9001  FORMAT('OPEN ERROR IN PRINT; #',I4)
89          RETURN
90    9100  WRITE(6,9101) ERRFLG
91    9101  FORMAT('READ ERROR IN PRINT; #',I4)
92          RETURN
93    9200  WRITE(6,9201) ERRFLG
94    9201  FORMAT('CLOSE ERROR IN PRINT; #',I4)
95          RETURN
96          END
```

ISIS-II PL/M-8Ø V3.1 COMPILATION OF MODULE INITMD
OBJECT MODULE PLACED IN :F1:INITMD.OBJ
COMPILER INVOKED BY: plm80 :F1:INITMD.plm DEBUG DATE(10/12/78) PAGEWIDTH(78)


```
     1              INITMD:
                        DO;

                    $nolist

    16     1        FQØGO: PROCEDURE EXTERNAL;
    17     2        END FQØGO;

    18     1        DECLARE semaphore (3) ADDRESS EXTERNAL;
    19     1        DECLARE token (3) STRUCTURE(
                        msgShdr) EXTERNAL;

    2Ø     1        INIT: PROCEDURE PUBLIC;
    21     2        DECLARE i BYTE;

    22     2  (X)       CALL FQØGO;

                    /* initialize semaphores */

    23     2  (Y)       DO i=Ø TO 2;
    24     3            CALL RQSEND(semaphore(i),.token(i));
    25     3            END;

                    /* PROGRAM THE 8255 */

    26     2            OUTPUT(ØEBH)=92H;

                    /* TURN OFF ALL ALARMS */

    27     2  (Z)       OUTPUT(ØEAH)=Ø;

    28     2        RETURN;
    29     2        END;
    3Ø     1        END INITMD;
```


ASM8Ø.OV3 :F1:X2CFG.M8Ø DEBUG PAGEWIDTH(78)

ISIS-II 8Ø8Ø/8Ø85 MACRO ASSEMBLER, V2.Ø   X2CFG   PAGE   1


```
  LOC  OBJ       SEQ          SOURCE STATEMENT

                    1          NAME    X2CFG
                    2          CSEG
                    3          PUBLIC  RQRATE
 ØØØØ 2ØØØ          4 RQRATE:  DW      32
                    5          $NOLIST
                  36Ø          $LIST
                  361          $NOGEN
 ØØØØ             362 NTASK    SET     Ø
 ØØØØ             363 NEXCH    SET     Ø
 ØØØØ             364 NDEV     SET     Ø
 ØØØØ             365 NCONT    SET     Ø
                  366 ;
                  367 ;        BUILD INITIAL TASK TABLE
                  368 ;
                  369          STD     RQADBG,64,129,RQWAKE
                  426          STD     RQTHDI,36,112,RQOUTX
                  483          STD     RQPDSK,48,129,RQDSKX
                  540          STD     RQPDIR,48,13Ø,RQDIRX
                  597          STD     RQPDEL,64,131,RQDELX
                  654          STD     RQPRNM,64,132,RQRNMX
                  711          STD     RQAIH,34,133,RQAIEX
                  768          EXTRN   RQHD1
                  769          CONSTD  CNTROL,RQHD1,8Ø,CNSTK,81,CONTEX
                  882          STD     TIMER,64,2Ø,Ø
                  939          STD     TIMUPD,64,14Ø,Ø
                  996          STD     TIMER5,64,141,Ø
                 1Ø53          STD     STSINP,64,142,Ø
                 111Ø          STD     CHANGE,64,143,Ø
                 1167          STD     REPORT,8ØØ,144,Ø,18
                 1224   (AA)   STD     SCAN,8ØØ,144,Ø,18
                 1281 ;
                 1282 ;        ALLOCATE TASK DESCRIPTORS
                 1283 ;
                 1284          GENTD
                 1288 ;
                 1289 ;        ALLOCATE EXCHANGES
                 129Ø ;
                 1291          XCH     CONTEX
                 1295          XCH     FQØLOK
```

```
                              BB    1299        INTXCH  RQL5EX
                                    1305 ;
                                    1306 ;      BUILD INITIAL EXCHANGE TABLE
                                    1307 ;
                                    1308        XCHADR  RQDSKX
                                    1315        XCHADR  RQDIRX
                                    1322        XCHADR  RQRNMX
                                    1329        XCHADR  RQDELX
                                    1336        XCHADR  RQAIEX
                                    1343        PUBXCH  CONTEX
                              CC    1350        PUBXCH  RQL5EX
                                    1357        PUBXCH  FQØLOK
                                    1364        XCHADR  RQINPX



              LOC  OBJ       SEQ          SOURCE STATEMENT

                             1371        XCHADR  RQOUTX
                             1378        XCHADR  RQDBUG
                             1385        XCHADR  RQWAKE
                             1392        XCHADR  RQALRM
                             1399        XCHADR  RQL6EX
                             1406        XCHADR  RQL7EX
                             1413        XCHADR  STSLOK
                             1420        XCHADR  SETLOK
                             1427        XCHADR  DSKLOK
                             1434        XCHADR  BMPTIM
                             1441        XCHADR  TIMPOL
                             1448        XCHADR  PRTREQ
                             1455        XCHADR  CHRESP
                             1462        XCHADR  ANRESP
                             1469        XCHADR  MIN5EX
                             1476        XCHADR  TIMEEX
                             1483        XCHADR  RDRESP
                             1490 ;
                             1491 ;      BUILD CREATE TABLE
                             1492 ;
                             1493        CRTAB
                             1500 ;
                             1501 ;      BUILD DEVICE CONFIGURATION TABLE
                             1502 ;
                             1503        DCT     DØ,Ø,Ø,Ø
                             1544        DCT     D1,Ø,Ø,1
                             1585 ;
                             1586 ;      BUILD CONTROLLER SPECIFICATION TABLE
                             1587 ;
                             1588        CST     Ø,8ØH,5,RQL5EX,CONTEX
                             1604 ;
                             1605 ;      BUILD BUFFER ALLOCATION BLOCK
                             1606 ;
                             1607        BAB     3,BUFPOL
                             1627        END
```

PL/M-8Ø COMPILER   10/12/78   PAGE   1

ISIS-II PL/M-8Ø V3.1 COMPILATION OF MODULE CAMMOD
OBJECT MODULE PLACED IN :F1:CAM.OBJ
COMPILER INVOKED BY: plm8Ø :F1:CAM.plm DEBUG DATE(1Ø/12/78) PAGEWIDTH(78)


```
    1           CAMMOD:
                    DO;
                /* CONTROLLER TASK STACK */

    2     1         DECLARE CN$STK (8Ø) BYTE PUBLIC;

          DD    /* DFS INTERNAL BUFFER SPACE */

    3     1         DECLARE RQDBUF (7ØØ) BYTE PUBLIC;

                /* DFS STATIC BUFFER POOL */

    4     1         DECLARE BUF$POL (12ØØ) BYTE PUBLIC;


    5     1     END CAMMOD;
```

intel®

ARTICLE
REPRINT

AR-41

An Integral Real-Time Executive
For Microcomputers

COMPUTER DESIGN July 1977

9800579A

Single-board microcomputers offer hardware cost-effectiveness for implementing many real-time systems. A compatible, resident, real-time executive program provides savings in software development

An Integral Real-Time Executive

For Microcomputers

Kenneth Burgett and Edward F. O'Neil

Intel Corporation
Santa Clara, California

Single-board computers, or microcomputers, that contain central processor, read-write and programmable read-only memory, real-time clock, interrupts, and serial and parallel input/output all on one printed circuit board, have made feasible a whole spectrum of applications which previously could not be economically justified. These microcomputers have also opened up a range of applications where the high functional density of large-scale integration provides advantages over previous solutions such as hardwired logic or relatively expensive minicomputers. While microcomputers readily solve hardware requirements, software for single-board computer applications with real-time characteristics (which are in the majority) has until now been generated individually for each application.

The Intel RMX/80* Real-Time Multi-Tasking Executive simplifies real-time application software development, and at the same time furnishes capabilities optimized for the microcomputer environment. It provides the means to concurrently monitor and control multiple external events that occur asynchronously in real-time. The program framework allows system builders to immediately implement software for their particular applications, and to avoid specific details of system interaction.

Major functions of the executive include system resource access based on task priority, intertask communication, interrupt driven device control, real-time clock control, and interrupt handling. In combination, these functions eliminate the need to implement detailed real-time coordination for specific applications.

Previously, two alternative software approaches were used to solve microcomputer applications. First, many designers created their own operating executive, individually tailored for each application. Obviously, this approach was expensive and time-consuming. The second approach was to use a minicomputer executive which had been adapted to a microcomputer. Since this software was designed for a different processing environment and then "stripped down," it suffered from major inadequacies when executed on microcomputers. The alternative, RMX/80, has been designed specifically to provide a general-purpose real-time executive tailored to Intel SBC 80 and System 80 microcomputers.

Real-Time System Requirements

All software design approaches for use in real-time applications include capability for concurrence, priority, and synchronization/communication.

Concurrence—Real-time systems monitor and control events which are occurring asynchronously in the physical world. Microcomputer software does not know exactly when external events will occur; however, it must be prepared to perform the necessary processing upon demand, whenever the events actually do occur. Typically, interrupts are used to inform the microcomputer that an event has occurred. At interrupt time, system control software determines what processing to perform, as well as the relative sequence in which processing must take place.

---

*RMX/80™ is a registered trademark of the Intel Corp, Santa Clara, Calif.

Programs related to external events are processed in an interleaved manner based on interrupt occurrence and priority. For instance, one routine is executing when an interrupt activates, signaling that a higher priority event has occurred. At this point, the routine related to the priority interrupt is started, while execution of the less important routine is discontinued temporarily. When the more important routine is completed, or temporarily halted for some other reason, execution of the less important routine is resumed. In this manner, multiple programs execute concurrently in an interleaved fashion.

Priority—In a real-time environment, certain events require more immediate attention than others because of their significance within the physical world. Immediacy is relative to other processing, and is determined by application requirements. The concept of immediacy or priority, however, is common throughout all real-time microcomputer applications. In priority-based systems, the most important program (one that is not waiting for some physical or logical reason) is the one executing.

A classic illustration of program priority in real-time systems is found in the area of plant control. When the plant begins to fail in a nonrecoverable manner, it is imperative that the plant be shut down as quickly as possible. For this reason, shutdown processing takes priority over all other system demands. Software priority enforces this hardware concept of physical operational events.

Synchronization/Communication—Another common similarity in most real-time systems is the need for synchronization between various events in the physical world which are under microcomputer control. Synchronization is defined as the process whereby one event may cause one or more other events to occur. Communication is the process through which data are sent between input/output (I/O) devices or programs and other programs within the microcomputer system.

An example of the need for synchronization and communication is a microcomputer system for weighing and stamping packages. One part of the system weighs the package, calculates pricing, and releases the package onto a conveyor belt. Price and weight data are communicated to another part of the system which stamps the data onto the package after it arrives at a sensor station. Synchronization is demonstrated by the occurrence of one event—package arrival—causing another event—package stamping—to occur.

Compatible Benefits

To satisfy real-time microcomputer software requirements, the RMX/80 Real-Time Executive software (Fig 1) was designed. This program differs from existing software systems by offering capabilities directly related to the single-board microcomputer environment in which it operates. These capabilities have two major bottom-line benefits compared with equivalent minicomputer systems. First, the executive code is compact enough to allow a large number of real-time applications to be processed on a single microcomputer board. To accomplish this capability, its nucleus is optimized to reside in less than 2k bytes [ie, in a single 16k programmable read-only memory (p/ROM)], thereby allowing up to 10K of onboard memory for application-related software and storage.
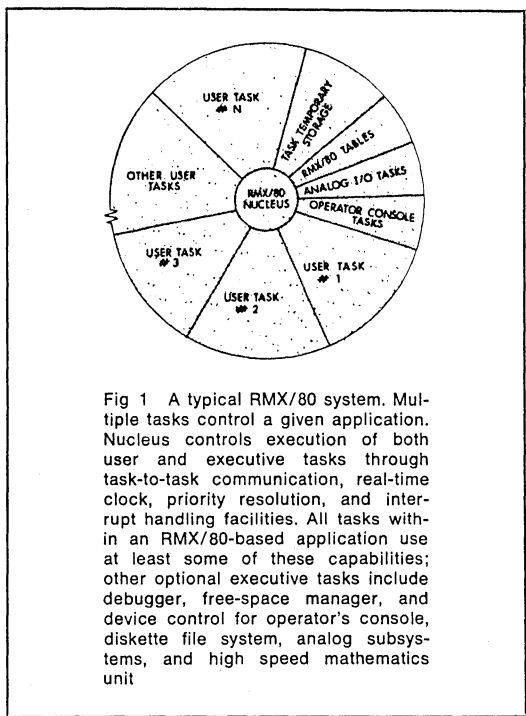
Fig 1   A typical RMX/80 system. Multiple tasks control a given application. Nucleus controls execution of both user and executive tasks through task-to-task communication, real-time clock, priority resolution, and interrupt handling facilities. All tasks within an RMX/80-based application use at least some of these capabilities; other optional executive tasks include debugger, free-space manager, and device control for operator's console, diskette file system, analog subsystems, and high speed mathematics unit

Second, the executive may be p/ROM-resident. When the microcomputer system is powered on, the software system (executive plus application programs) is automatically initialized and begins execution of the highest priority application task. Typical major real-time executives, however, are totally random-access read-write semiconductor memory (RAM)-resident, which means they must be initialized (booted) from a peripheral device, such as diskette, cassette, or communications line, into microcomputer memory. The need for peripheral devices significantly increases the total cost of traditional real-time executive-based solutions.

Sample Application

Functioning as a real-time executive for microcomputers, this software system provides facilities for orderly control and monitoring of asynchronously occurring external events. Although these events may differ widely from application to application, facilities are adaptable to nearly all processes where the microcomputers are used, including process and machine control, test and measurement, data communications, and specialized on-line data processing applications (where one or more terminals access diskette-based data). The executive is particularly useful in dedicated low cost applications which were not economically feasible before the advent of microcomputers. For example, consider the requirement of gas pump control in a service station (Fig 2).

In this station, a microcomputer system operating with RMX/80 concurrently monitors and controls multiple gas pumps, and sends price and volume informa-

tion to one central location. At the same time, information about station operation is being transmitted over a communications line to a regional computer.

Individual tasks are developed independently to measure gas flow, calculate and display price information, transfer data to the central computer, and monitor levels of gasoline in underground storage. All this processing takes place concurrently under program control. (Credit verification, charge slip printing, and billing can also be controlled by additional software tasks.)

Efficient gas station operation demands that the hardware/software system be highly reliable. The compatible benefits of compact code, p/ROM residency, and self-initialization on a single-board microcomputer system all combine to ensure functional integrity.

Software Structure

RMX/80 simplifies the effort for developing a real-time system, first, by providing many commonly required software functions. Second, its software structure promotes efficient program development. Programmers who are familiar with structured programming will find task orientation both natural and easy to use.

Tasking means that a larger program is divided into a number of smaller, logically independent programs or tasks. The key is to identify functions that may occur concurrently. For example, consider the tasks required for a terminal handler—real-time asynchronous I/O between an operator's CRT terminal and the executive.

Input Handler Task—One task must be ready to accept a data character from the terminal at any time. This is done by responding to an interrupt signal from the terminal and then accepting the data character. The task immediately passes the input character to a subsequent task automatically and then goes back to wait for another interrupt.

Line Buffer Task—As characters are received from the input handler they must be placed into a buffer to form a line. Eventually, the buffer will be filled or the logical end-of-line will be signaled by a carriage return character. At this point, the line buffer must be sent to some other task for processing.

Echo Driver Task—For a full-duplex terminal, it is necessary to return each input character to the terminal for display on the CRT screen. This task waits for a character, which could be sent by either the line buffer or input handler task, and then sends the character to the terminal. It then waits for the next character.

Note that input handler and echo driver are described as waiting for an event. Within the RMX/80, that is literally the case. While they wait, however, system resources are available for other tasks, such as that of the line buffer. Thus, effective processing may occur concurrently with necessary waiting periods. Notice also that a number of other tasks may also be active within the system. In fact, the greater the number of tasks running concurrently, the more effectively system resources are used. Concurrent operation eliminates many time wasting procedures from a real-time system. For example, the executive can eliminate the need for many timing loops where the processor simply executes a no-operation instruction repeatedly while waiting for an event to occur.
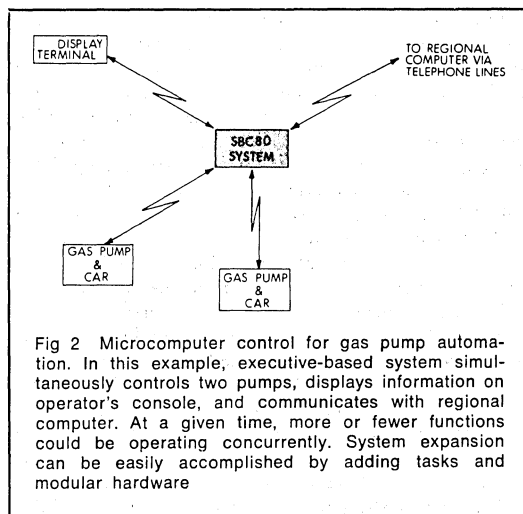
Fig 2 Microcomputer control for gas pump automation. In this example, executive-based system simultaneously controls two pumps, displays information on operator's console, and communicates with regional computer. At a given time, more or fewer functions could be operating concurrently. System expansion can be easily accomplished by adding tasks and modular hardware

Within the executive, tasks not only are logically independent, they are also physically independent, actually contending with each other for the use of the processor and other system resources. The executive resolves this contention based on the priority of each task.

In the terminal handler example, it is clear that the input handler must have highest priority, since acceptable performance cannot tolerate the loss of data. Second highest priority is given to the echo driver, so that data appearing on the screen remain coordinated with the input. Lowest priority goes to the line buffer, since that function does not depend directly on an external asynchronous event. There are no particular real-time constraints on the line buffer as long as the input characters are eventually processed.

It is possible to write the entire terminal handler as a single large task instead of as several smaller tasks. However, consideration must be given other high priority tasks operating within the system which may not be able to gain control while a low priority portion of the terminal handler, such as the line buffer task, is executing. Therefore, tasks assigned as high priority are generally kept as short as possible. If the terminal handler were written as one large task, it could tie up the entire processing system for a relatively trivial function.

Task States

Two task states have been implied—running and waiting. A running task is always the task which currently has the highest priority and is not suspended or waiting. A waiting task remains in the wait state until it receives a message or an interrupt for which it is waiting or until a specified time period has passed. The wait period can be timed using the system clock.

A running task may suspend itself on some other task in the system. A suspended task cannot begin execution again until some running task orders it to resume. As an example, a password routine might temporarily suspend the echo driver of the terminal handler so that the password is not displayed. (The password routine must
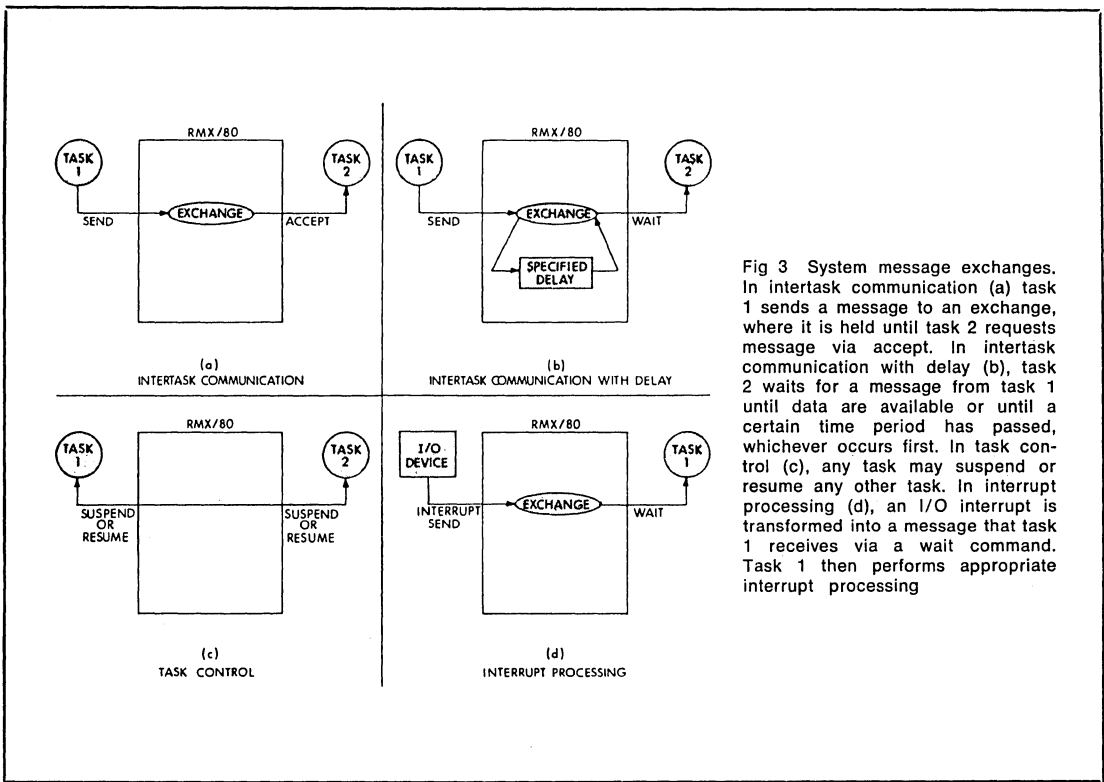
Fig 3 System message exchanges. In intertask communication (a) task 1 sends a message to an exchange, where it is held until task 2 requests message via accept. In intertask communication with delay (b), task 2 waits for a message from task 1 until data are available or until a certain time period has passed, whichever occurs first. In task control (c), any task may suspend or resume any other task. In interrupt processing (d), an I/O interrupt is transformed into a message that task 1 receives via a wait command. Task 1 then performs appropriate interrupt processing
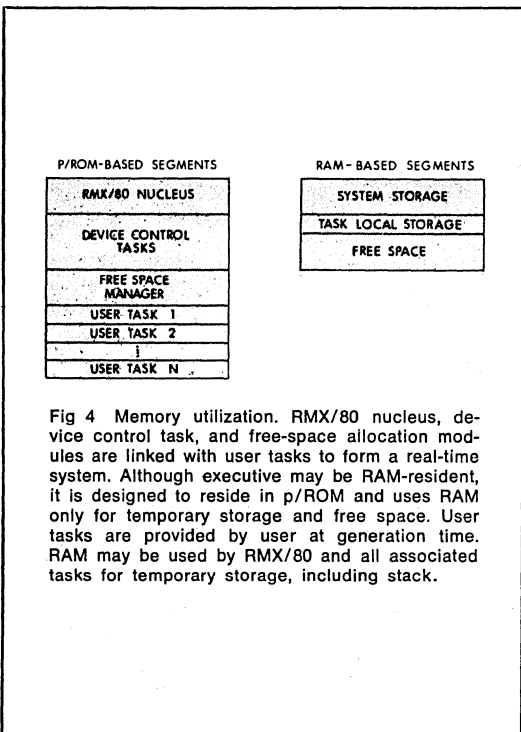
Fig 4 Memory utilization. RMX/80 nucleus, device control task, and free-space allocation modules are linked with user tasks to form a real-time system. Although executive may be RAM-resident, it is designed to reside in p/ROM and uses RAM only for temporary storage and free space. User tasks are provided by user at generation time. RAM may be used by RMX/80 and all associated tasks for temporary storage, including stack.

remove the password from the line buffer, or it will be displayed as soon as execution of the echo driver is resumed.)

A task may also be in the ready state. A ready task is one that would be running except that a task with higher priority temporarily controls the system resources. The executive maintains a list of all tasks that are ready to run. The next task to be run is always the task with the highest priority in the ready list.

The running task relinquishes its control of the system by

(1) Putting itself into a wait state

(2) Suspending itself

(3) Sending a message to a higher priority task, which if it has the highest current priority, becomes the running task

(4) Being preempted by an interrupt to a higher priority task

In the case of an interrupt, the executive saves the status (contents of registers, etc) of the interrupted task so that it will be restarted correctly.

Message Exchanges

Tasks communicate with each other by sending messages (Fig 3). The sending task constructs the message to be sent in RAM or uses a previously assembled message.

TASK ENTRY POINT

INITIALIZE TASK

WAIT FOR REQUEST

PERFORM FUNCTION
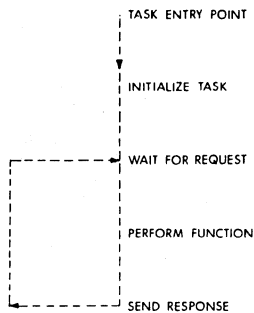
SEND RESPONSE

Fig 5 Consumer task flow. Consumer task performs initialization and then drops into cyclic loop, alternately waiting for messages, performing functions requested by message, and sending an acknowledgement in form of a response message

TASK ENTRY POINT

INITIALIZE TASK

PERFORM FUNCTION

INITIALIZE OPERATION (SEND MESSAGE)

WAIT FOR RESPONSE
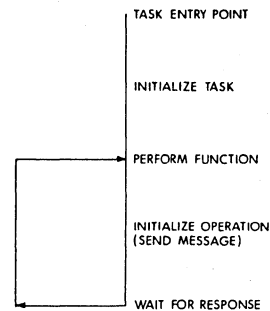
Fig 6 Producer task flow. Producer processing flow is opposite to that of consumer task. Instead of passively reacting to requests from other tasks, producer task issues requests to which other tasks must respond

The sending task then issues a SEND command that posts the address of the message at an exchange.

An exchange is simply a set of lists maintained by the executive. The first list contains the addresses of messages available at that exchange. The second list consists of a list of tasks that are waiting for messages at that exchange. When a task enters a wait state, it specifies the exchange where it expects eventually to find a message. The task may wait indefinitely, or it may specify that it will only wait a specific period of time before resuming execution.

Messages, together with the exchange mechanism, provide for automatic intertask communication and also for task synchronization. For example, a message to a particular task may specify that the task is to send a response to a certain exchange. Thus, the original task may request an acknowledgement response to its message, or it may specify that a message is to be sent to a third task. RMX/80 treats interrupts like messages, the only difference being that interrupts have their own set of exchanges.

Note that the sending and receiving of messages classifies tasks into two types—message consumers and message producers. A consumer task waits for a message, performs an action based on the message, and then returns to the wait state until another message is received. A producer task initiates its function by sending a message to another task, waits for a response, and then sends another message. Figs 5 and 6 graphically illustrate the processing within these two tasks. The distinction be-

tween consumer and producer tasks is relative since many tasks act as both consumer and producer.

Executive Modules

RMX/80 is supplied as a library of relocatable and linkable modules. These modules are added selectively as required when the user-supplied tasks are passed through the link program. Only modules actually requested by the application are linked in. For example, if the application program does not specify use of the free-space manager, that module is not linked into the system.

One module, the nucleus, provides basic capabilities (concurrence, priority, and synchronization/communication) found in all real-time systems. Additional, optional modules may be configured with user programs (tasks) to form a complete application software system. These modules include:

Terminal handler—Providing real-time asynchronous I/O between an operator's terminal and tasks running under the RMX/80 executive, the handler offers a line-edit feature similar to that of ISIS-II and an additional type-ahead facility. (ISIS-II is the supervisory system used on the Intellec Development System.)

Free-space manager—This module maintains a pool of free RAM and allocates memory out of the pool upon request from a task. In addition, the manager reclaims memory and returns it to the pool when it is no longer needed.
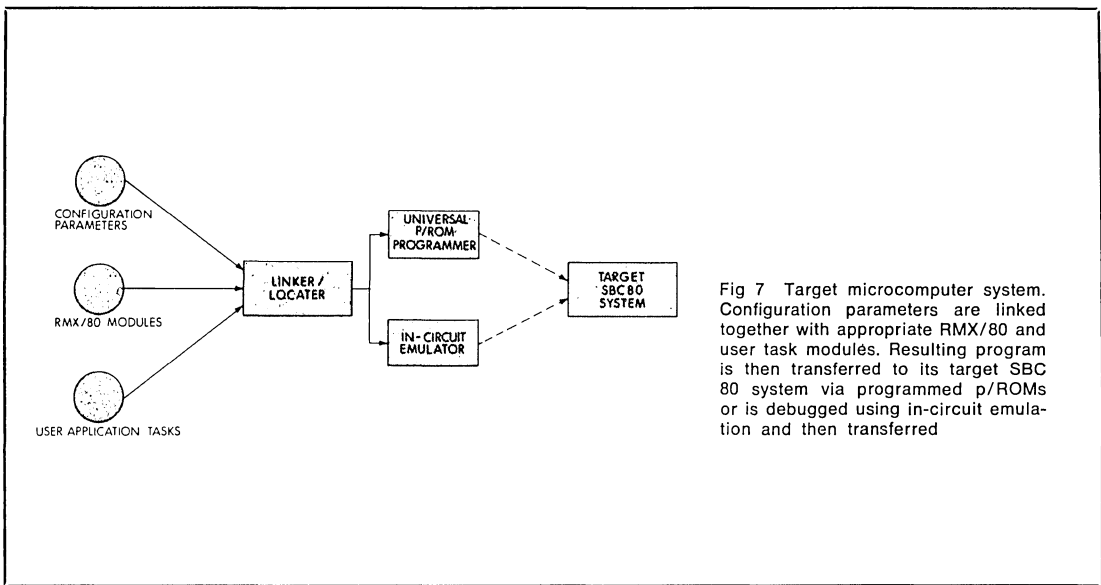
Fig 7 Target microcomputer system. Configuration parameters are linked together with appropriate RMX/80 and user task modules. Resulting program is then transferred to its target SBC 80 system via programmed p/ROMs or is debugged using in-circuit emulation and then transferred

Debugger—Designed specifically for debugging software running under the RMX/80 executive, the debugger is used by linking it to an application program or task. Thus, it can be run directly from the single-board computer's memory. In addition, an in-circuit emulator, such as ICE-80, can be used to load and execute the debugger, providing all resources of the Intellec development system to simplify debugging effort.

Analog interface handlers—Consisting of RMX/80 tasks, these handlers provide real-time control for SBC 711, 724, and 732 systems.

Diskette file systems—Giving RMX/80 users diskette file management capabilities, the diskette driver allows users to load tasks into the system and to create, access, and delete files in a real-time environment without disrupting normal processing. All file formats are compatible with ISIS-II for both single and double density systems.

In addition to application program module or task requirements, the user also supplies a set of generation parameters. These parameters are a set of tables that inform the executive of the number of tasks and exchanges in the system. Fig 7 illustrates the system generation process.

Summary

The significance of RMX/80 to software design parallels the significance of the single-board computer to hardware design. Microcomputers allow designers without extensive experience in digital systems to bring computer processing power into their applications. Similarly, the executive relieves the hardware designer of much software design required for real-time applications. Designed to facilitate growth, since new software needed to support hardware expansions can be supported easily by the addition of new tasks, this executive also substantially re-

duces recurring costs because it requires a minimum of memory and does not require peripheral bootstrap loading devices. RMX/80 results in economical, shorter, and more flexible software development efforts when designing, building, and verifying real-time user applications.

Bibliography

C. G. Bell, A. Newell, *Computer Structures: Readings and Examples*, McGraw-Hill, New York, 1971

P. Brinch-Hansen, *Operating Systems Principles*, Prentice Hall, 1973

E. W. Dijkstra, "The Structure of the *THE* Multiprogramming Systems," *Communications of the ACM*, May 1968, pp 341-346

E. I. Organick, *The Multics System: An Examination of Its Structure*, MIT Press, Cambridge, Mass, 1972

D. M. Richie, K. Thompson, "The UNIX Time Sharing System," *Communications of the ACM*, July 1974, pp 135-143

A Small-Scale Operating System Foundation for Microprocesor Applications

KEVIN C. KAHN

Abstract—Sound engineering methodology, which has long been valued in hardware design, has been slower to develop in software design. This paper uses a case study of a small real-time system to discuss software design philosophies, with particuar emphasis on the abstract machine view of systems. It demonstrates how the currently popular software design axioms of generality and modularity can be used to produce a software system that meets severe space constraints while remaining relatively portable across a family of microcomputers. These sorts of constraints have often been used to justify *ad hoc* design approaches in the past. The results of the project suggest that the use of such techniques actually make the meeting of many constraints easier than would a less organized approach. In addition, the reliability and maintainability of the resultant product is likely to be better.

I. Introduction

A PROCESSOR, as defined only by its hardware, is typically not an adequate base upon which to build applications software. Broad classes of applications can be examined and found to share more than the hardware defined instruction set. To avoid the reengineering of this common functionality, we would prefer to build such common parts once and thereafter treat this base software as though it were part of the machine. For example, a software system sometimes called an operating system, an executive, a nucleus, a kernel, or some similar term, is often supplied with a hardware product and can be viewed in exactly this way. In this paper, we examine a small-scale system to demonstrate this approach to bridging the gap between the hardware and the application. That is, we will view the software as a direct extension of the hardware—a view which may indicate future directions in microprocessor integration of function.

This paper is meant as both a case study of a particular system design and as a suggestion of the proper approach to such design situations in general. We will first discuss the abstract machine view of computer systems and attempt to demonstrate that this is a useful philosophical approach for building systems. We will then apply this approach to the discussion of a system to coordinate programs performing real-time control functions— RMX-80™ [18]. The emphasis of the paper will be on techniques and methodology rather than on the particular functionality of RMX. Special attention will be given to such issues as the use of modularity to enhance the adaptability of the system and the use of design generality to achieve global rather than local optimizations.

II. The Concept of Abstract Machine

What is a computing "machine" or processing unit? We generally identify a processing unit as a particular collection of hard-

ware components that implement the instruction set of the machine. This very physical definition of a computer dates from mechanical processors. Even with modern computers, before large-scale integration, it was easy to physically point at the processing elements as distinct from memories, peripherals, and programs. Continued integration of function has at least made this physical distinction more difficult with single chips subsuming processing, memory, and peripheral interface functions. Microprogramming (i.e., replacing hardwired instruction logic with a more elementary programmed processor) as an implementation strategy has logically blurred this distinction as well. That is, when the basic visible instruction set of a processor is itself implemented in terms of more primitive instructions it is more difficult to identify "the machine." It is clear that this narrow physical definition of a processor is not adequate for current technology levels and is likely to become even less viable as the technology continues to develop.

Actually we have been using alternative definitions of a processor for some time. All of the theoretical work in finite state machines, for example, deals with conceptual processors. Likewise applications programmers seldom really regard the machine they program as much more than collection of instructions found in a reference manual—the physical implementation of the machine is of little concern to them. Indeed, they may never come physically near the hardware if they deal with a typical time-sharing system—rather, the terminal is the only physical manifestation of the computer such users may see.

More to point, perhaps, are the numerous interpreters that have been written for languages such as Basic. Each such interpreter actually produces a conceptual machine with one instruction set targetted to a specific application. With standard compiled languages such as Fortran, Algol, or Pascal, a higher level source statement is translated into the instruction set of the physical hardware. In contrast, interpreted language systems translate the source into the instruction set of some conceptual machine that is better suited to the running of programs written in the language. For example, the hardware may not provide floating-point instructions or define a floating-point data representation. In such a case it may be easier to define a machine that recognizes a particular floating-point data format with an instruction set that includes floating operations. These interpreters are high-level machines that have usually been implemented in software. Likewise, it should be readily apparent that, just as these interpreters provide high-level machines to their associated translators, any programming language, compiled or interpreted, provides one to its users.

Interpreters of this sort typically may examine and decode a stream of instruction values in a manner analogous to the hardware. Alternately, the new instructions may all be executed as subroutine calls using the appropriate hardware instruction. That is, the entire bit pattern for CALL X (where X is the address of a
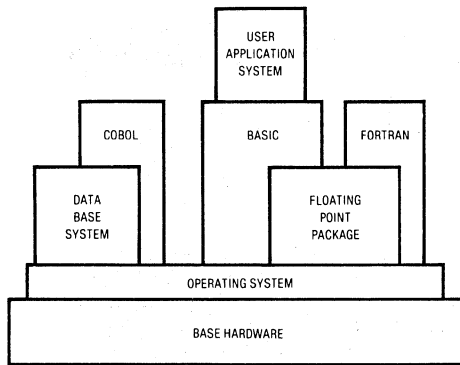
Fig. 1. Typical collection of abstract machines.

routine that implements a part of the new instruction set) can be regarded as a new operation code rather than as the hardware operation CALL. In either case the programmer using these extensions can view the harware–software combination as though it were a new machine with a more useful instruction set. Microprogrammed machines such as the IBM 5100 or Burrough's 1700 have simply optimized the performance of such interpreters or subroutine packages by committing them to a faster storage medium.

Viewed in this light we can identify any collection of hardware and software that provide some well defined set of functions as defining an *abstract machine* [10],[12]. This machine has an instruction set that consists of the functions provided by the hardware–software combination. For a particular application it may be possible to view multiple such abstract machines by taking various pieces of the whole. For example, the physical machine provided by a set of components is just one abstract machine. It is of particular interest since it is the greatest common abstract machine that can be identified as being used by any application running on that computer system. A Basic interpreter running on this machine might then constitute a second virtual machine. A Basic program running on this interpreter that accepted high level commands and performed according to them might be a third level machine usable by people with no knowledge of either the hardware or Basic. Whenever we can identify functions of sufficient commonality among a number of applications, it may be worth viewing the software which provides these functions as extensions of the base hardware machine which define some new augmented or even different machines. Users programming such an application can then view this abstract machine, rather than the base machine, as the vehicle that they are programming, and in doing so avoid reengineering the functions that it provides. Fig. 1 illustrates an example of such machines. It is important to remember that at any time, many abstract machines may be thought of as existing on the same base hardware.

III. OPERATING SYSTEMS AS ABSTRACT MACHINES

The terms operating system or executive have been used to describe software systems of widely different functionality. These machines generally provide for the management of some machine resources such as input, output, memory space, memory access, or processor execution time. We might then attempt to define an operating system as some collection of software modules which defines an abstract machine that includes resource management functions as well as the hardware supplied computational func-

tions [2],[6],[8],[11]. With such a broad definition, however, large-scale multi-user time-sharing systems and small single user microprocessor development systems both may claim to have operating systems. Clearly, the range of software systems covered by this definition is large, encompassing products which differ by orders of magnitude in complexity. Rather than become involved in trying to resolve this disparity, we will qualify our use of the term and refer to an operating system "foundation." That is, we will describe a software system which provides a minimal base for the construction of real-time applications. We will avoid the somewhat irrelevant question of whether the system comprises a complete "operating system."

The important item to realize from the above discussion is that any operating system functionally enlarges the processor seen by the programmer. The functions that it provides become as much a part of the machine's functionality as jump instructions. Indeed, it is functionally unimportant to the user desiring to read from a file whether it requires a single hardware instruction or a large software routine to accomplish it. In terms of the abstract machine discussion above, we will examine a software package which defines an abstract machine that includes functions required to coordinate programs performing real-time control applications [1],[9],[12].

The key overall requirement of the operating sytem foundation that we discuss in this paper will be that it supply a minimal covering set of functions to permit coordination of asynchronous tasks. To determine this set we will need to further examine the needs of its users and environment of its use. In describing this foundation, we are defining an abstract machine that must be programmed to be of use; that is, like the instruction set of the base machine the foundation by itself performs no work but rather provides an environment within which useful tasks can be run.

We should note, here, some of the limitations of the system which differentiate it from large-scale operating systems. First, it is not primarily intended for a multi-user environment, particularly because the underlying hardware does not provide the necessary support to protect users from one another. Also, it will often be used to control functions of specialized devices and therefore is "close" to the I/O devices. That is, it does not supply the sort of high level I/O control system which is often present in larger systems for controlling more conventional I/O devices. Finally, it does not assume a backing store from which program overlays can be loaded (but it can easily support such an extension).

IV. DESIGN CONSIDERATIONS

A. Use Environment

The foundation system we will describe is RMX-80 [5] which was designed to be used with members of Intel's Single Board Computer (SBC) family of products. This family includes a wide range of bus compatible processor, memory, and peripheral boards. Of most interest to this discussion are the processor boards which are based on the Intel 8080 or 8085 microprocessors and include varying amounts of on-board ROM and RAM memory and I/O interfaces. In addition, the boards vary in the sophistication of their interrupt structures and timing facilities. In terms of abstract machines we might characterize these computers as essentially the same machine at the processor level but different machines at the computer system level. It was desired that the abstract machines defined by adding RMX to the underlying computers be as much the same as possible.

During the design of RMX, we expected that its users would span the entire broad range of applications across which the SBC

hardware was being put to use. This implied that it might see uses ranging from minimal single board systems that functioned as single device controllers to complex multiboard applications implementing involved real-time process or industrial control functions. In particular we expected that many user-built I/O boards and peripherals would be used with the system. It was important for us to allow full use of these unknown devices with RMX while still providing as much assistance as possible in the building of the controlling software systems.

As is the case with most processors, the concrete (i.e., physical) machines represented by the SBC family do not themselves include any facilities to permit multiple asynchronous functions to be programmed, to provide for the coordination of such functions, or to provide time information needed for real-time applications. Typically, users of these products have directly programmed these functions in an *ad hoc* manner within their applications. An examination of the sorts of functions necessary to such applications reveals that at the very least this reengineering is a waste of resources. Worse is the high probability of error in programming such critical functions.

The SBC hardware products were designed to eliminate the complexities of board engineering, particularly for those users without the necessary expertise to handle the task, by functionally integrating individual components into complete boards. The programming of functions to coordinate parallel software activities is, likewise, an area which should be carefully engineered in order to avoid subtle errors. The development of RMX was therefore viewed as a process of functional integration analogous to the integration of LSI components into boards. That is, just as a well designed board relieves the user of component level hardware engineering, RMX relieves the users of low-level software engineering.

B. System Requirements

The hardware environments and anticipated uses of RMX defined a stringent set of requirements for it. Foremost among these were its memory constraints; indeed, for the anticipated uses, memory size considerations dominated execution speed ones over a considerable range. Since we expected applications that would reside entirely on a single board with 4K bytes of PROM, the maximum size of the RMX foundation code was set at half of this or 2K bytes. Further, unlike larger minicomputer systems, many, if not most, applications of the SBC boards would not have available any mass storage or other program loading device. It was thus important that RMX be designed to be ROM (or PROM) resident and capable of automatically initializing the system when powered on.

We also anticipated that the expertise of many RMX users would be in areas other than programming systems. We therefore felt that the RMX machine needed to provide a fairly simple set of concepts, avoiding where possible those constructs most likely to cause errors. For example, we felt that a very frequent source of programming difficulty lay in dealing with interrupts. Many latent errors in programming systems stem from the occurrence of an interrupt at an unexpected time. We therefore decided to attempt to minimize the need for users to deal with hardware interrupts or with the interrupt-like occurrences found in many minicomputer operating systems. At the same time we had to accommodate the needs of the sophisticated user who still desired to take advantage of RMX but had a specific need to directly control the hardware via the interrupt facility.

Finally, to define the general functionality of RMX we examined its anticipated applications. Real-time applications commonly need to perform a number of tasks of differing importance

logically in parallel, with preference always being given to executing the most critical ones first. While these tasks may be relatively independent, they may need to periodically synchronize themselves with one or another distinct task or with the outside world. For the latter, interrupts are the usual hardware supplied mechanism. Some tasks may also need to communicate data with one another. For example, a task servicing a sensing device may take readings from the device which need to be communicated to two tasks: one task which reacts to the reading by controlling some other device, and another task which logs or tabulates the readings. Ranked in order of importance these might be control, sensing, and logging. Finally, the tasks must have the ability to control themselves relative to real-time, either by delaying their execution for certain periods or by guaranteeing that they are not indefinitely delayed by, for example, a faulty device.

Requirements on the system design were also generated by considerations internal to the design project. One of these was the need to provide a single RMX abstract machine on a variety of underlying SBC boards. While separate versions of RMX for each board could have been designed with the same external appearance, this approach would have led to an unnecessary amount of internal engineering. Additionally, without careful initial design, the differences in the base hardware would have had visible effects on the RMX abstract machine for each of the boards. This requirement demanded that we partition the structure of RMX into two parts. One part would implement those aspects which were independent of the particular hardware. The second part would interface the first part to the underlying hardware of the specific boards [7].

We also wished to minimize the software development costs by applying the best available software engineering techniques. Historically, tight space constraints have often led to a very *ad hoc* approach to software design in the belief that more generally designed external features or more modularly built internal designs would lead to inherently larger systems. As a result of this philosophy, each needed function is designed to be as small as possible. Unfortunately, while each function may be locally optimized, it is possible that the overall design suffers from duplication or overlap between such individual elements. Current work in programming methodology stresses modularity, generality, and structure (most often for their side effects in producing more maintainable, less error prone systems).

We felt that there was more to gain, both in development cost and space performance, by avoiding optimized specialization of function in favor of more general designs [17]. This reduced the number of separate functions that RMX had to supply. The resulting external design therefore has a single mechanism that provides task communication, synchronization, time references, and standard interrupt-like control. To do so it incorporates the operating system design approaches favored in much of the modern computing literature. Likewise, the internal structures are highly modular and designed to be as uniform as possible so as to avoid replicating similar, but nonidentical internal management routines.

V. The RMX Machine

B. General Concepts

The abstract machine defined by RMX augments the base microprocessor by introducing some additional computational concepts. We define a *task* to be an independently executable program segment. That is, a task embodies the concept of a program in execution on the processor. RMX permits multiple tasks to be defined which can run in a parallel, or multiprogrammed,

fashion. That is, RMX makes individual tasks running on one processor appear to be running on separate processors by managing the dispatching of the processor to particular tasks. The registers on the processor reflect the activity or state of the running task. Other tasks may be ready to execute but for some reason have not been selected to run yet and so have their processor states saved elsewhere in the system. From the point of view of the program that is a task, execution proceeds as though it were the only one being run by the processor. Only the apparent speed of execution is affected by the multiprogramming. From the point of view of the system, every task is always in one of three states: running, ready, or waiting. The task actually in execution is running. Any other task which could be running but for the fact that the system has selected some other task to actually use the processor, is ready. Tasks which are delayed or stopped for some reason are waiting, as will be discussed below.

Each task is assigned a *priority* which determines its relative importance within the system. Whenever a decision must be made as to which task of those that are ready should be run next, the one with the highest priority is given preference. Furthermore, in the spirit of unifying mechanisms, the same priority scheme replaces a separate mechanism for disabling interrupts. Interrupts from external devices are logically given software priorities. If the applications system designer deems a particular task as of more importance than responding to certain interrupts, he can specify this by simply setting the RMX priority of that task to be higher than the RMX priority associated with those given hardware interrupts. It is thus possible to maintain a high degree of control over the responsiveness required for various functions.

As mentioned above, tasks may desire to communicate information to one another. To this end the RMX machine defines a *message* to be some arbitrary data to be sent between tasks. To mediate the communication of messages it defines an *exchange* to be the conceptual link between tasks. An exchange functions somewhat like a mailbox in that messages are deposited there by one task and collected by another. Its function is complicated by the fact that a task may attempt to collect a message at an exchange that is empty. In such a case the execution of that task must be delayed until a message arrives. Tasks that are so delayed are in the waiting state. We chose this indirect communication mechanism over one which directly addresses tasks because it permits greater flexibility in the arrangement of receiver and sender tasks. The anonymity of the receiving task implies that the sender need know only the interface specification for a function to be performed via a message to a particular exchange. The task or tasks which implement that function need not be known and hence may be conviently changed if desired.

The conventional mechanism used by programs to communicate with external devices is the interrupt. Unfortunately, interrupts are by nature unexpected events and programming with them tends to be error prone. The essential characteristic of an interrupt is that a parallel, asynchronous activity (the device) wishes to communicate with another activity (a program). Since this communication is essentially the same as that desired between separate software tasks it seems conceptually simpler to use the same message and exchange mechanism for it. The unification of all communications functions is analogous to the idea of standardized I/O found in systems such as UNIX [17]. The RMX machine eliminates interrupts by translating them into messages which indicate that an interrupt has occurred. These messages are sent to specific exchanges associated with particular interrupts. Tasks which "service interrupts" do so in RMX by attempting to receive a message at the appropriate exchange. Thus, prioritized nested interrupts are easily handled. An advantage of this unfied

treatment of internal and external communication is that hardware interrupts can be completely simulated via another software task. This facilitates debugging and permits easy modification of a system by allowing rather arbitrary insertion of tasks into a network of communicating tasks and devices.

Note that with this scheme unexpected interrupts do not cause particular difficulty. For example, if the servicing task is still busy with some previous message, the interrupt message will be left at the exchange and will not affect the task until it is ready for another interrupt; i.e., until it waits at the exchange. In an application designed to properly handle the actual interrupt rate, the task will service interrupts quickly enough to always be waiting when the next one occurs. In this case, response to an interrupt is immediate. Thus this mechanism provides no loss of facility relative to the usual interrupt scheme but it does make the proper controlling of such events simpler. Multiple occurrences of the same interrupt which indicate the processor has fallen behind in its servicing are logged as such by a message which indicates that interrupts may have been lost. These interrupts do not, however, disrupt the running task or complicate programming.

The last concept embodied in the RMX abstract machine is that of time. The RMX machine defines time in terms of *system time units*. It then permits tasks to delay themselves for given periods of time so that they can synchronize themselves with the outside world. It also permits tasks to guard against unduly long delays caused by attempting to collect a message at an empty exchange by limiting the length of time that they are willing to spend waiting for some message to arrive.

B. Data Objects and Functions

These concepts are realized in RMX by introducing some new data objects and instructions. Just as the base processor can deal directly with such data objects as 8 bit bytes or unsigned integers, the RMX abstract machine can deal directly with the more complex data objects: task, message, and exchange. Each of these data objects consists of a series of bytes with a well defined structure and may be operated upon only by certain instructions. This is completely analogous, for example, to a machine that permits direct operations on floating-point data objects which consist of four bytes with a particular internal structure to represent the fraction, exponent, and signs. In each case there are only certain instructions that can be used correctly with the object and the internal structure of the object is not of particular interest to the programmer.

The new instructions provided by RMX are: SEND, WAIT, ACCEPT, CREATE TASK, DELETE TASK, CREATE EXCHANGE, and DELETE EXCHANGE. The create instructions accept blocks of free memory and some creation information to format and initialize the blocks with the appropriate structure. Each corresponding delete instruction accepts one of the objects and logically removes it from the system. The remaining operations are of more direct interest to the operation of the RMX machine.

The WAIT instruction has two operands: the address of an exchange from which a message is to be collected and the maximum time (in system units) for which the task is to await the arrival of a message. The result of the operation is the address of the message which was received. A special message from the system indicates that the specified amount of time elapsed without the arrival of a normal message. From the programmer's point of view this instruction simply executes and returns the specified result. Actual execution of the instruction will involve the delaying of task execution if no message is available, by queueing it in a first-come-first-served manner at the exchange. Any such delay is not visible

to the programmer, however. This approach unifies the communication and timing aspects of the design. It directly provides reliability in the face of lost events due to hardware or software failure. Tasks can be guaranteed not to be indeterminately delayed due to such failures and can thus attempt recovery from them. It also permits tasks to use the same mechanism to delay themselves for given time intervals by waiting at an exchange at which no message will ever arrive.

The ACCEPT instruction is an alternate way to receive a message. It has a single operand specifying the exchange from which the message is to be received and immediately returns either the next message at the exchange or a flag indicating that no message was available. The task is never delayed to await a message in the ACCEPT operation.

SEND also has two operands: the address of a message and the address of an exchange to which the message is to be sent. The instruction queues the message in a first-come-first-served manner at the exchange if there is no task already waiting there. If a task is waiting at the exchange then the instruction binds the message to the task and makes the task eligible to execute on the processor. When the receiving task resumes actual execution the address of the message will be returned to it as the result of its WAIT instruction.

VI. THE RMX IMPLEMENTATION

A. Methodology

In this section and the next, we consider some (but certainly not all) details of the actual implementation of the system as illustrations of the design of such software products. We turn first to the methodology applied to the effort and then to some samples of the mechanisms.

To provide the abstract machine just described and meet the other requirements for the system, RMX was implemented as a combination of ROM resident code and some RAM resident tables. Just as a hardware designer uses LSI devices in preference to more elementary TTL components, we chose to use the leverage of a high level programming language rather than elementary assembly code. The system was, therefore, designed using PLM [14], Intel's high-level implementation language. The operations described above appear as procedure calls using the standard PLM calling sequence. The space constraints and a good level of internal maintainability were achieved by maximizing the modularity of the design. The broad independent functions of multiprogramming, communications and control were completely isolated from the board dependent timing and interrupt handling functions. As a result, movement of the system to a new member of the SBC family requires only the reimplementation of these board dependent functions. In addition, data structure of internal and user visible objects were generalized so that single algorithms could deal with any of them. Individual optimizations could have been made in the local design of many parts of the data structures to improve their space or time costs slightly. Such optimizations, however, would have cost considerably more in code space and code complexity [3].

The module feature of PLM was used to simulate the abstract data type concept [4],[13] and enforce information hiding [15], [16]. That is, every data structure used by RMX is under the exclusive control of a single module. The modules supply to each other restricted sets of public procedures and variables. It is only through these paths that agents outside a module may access the data structures maintained by the module. The only assumptions that such outside agents may make about a module and its data structures are those specified by the definition of the public paths.
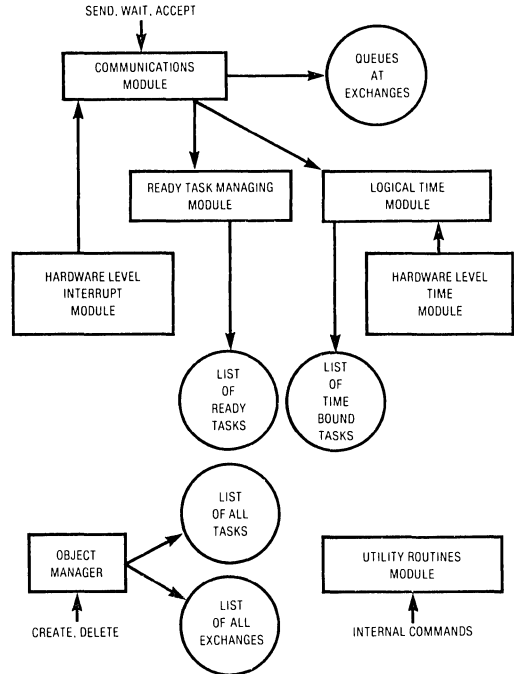
Fig. 2. Major modules (boxes) and data structures (circles) of RMX.

As a result, so long as these interface specifications are maintained, any given data structure may be reorganized by redesigning its controlling module without affecting other parts of the system. This approach improves the understandability of the implementation and facilitates the debugging and maintenance of the system. Fig. 2 illustrates the general structure of the RMX implementation.

Finally, the original version of RMX was completely coded in PLM using the resident PLM compiler of the Intellec® Microcomputer Development System. This version was functionally complete but slightly exceeded the space constraints, occupying about 2.5K bytes of program space. There were a couple of cases where the language structure of PLM did not permit the direct expression of the best way to compile the code. For these modules, it was sufficient to hand optimize the code output by the compiler. The original structure of the PLM program was maintained and the majority of its generated code was used intact. The final RMX system occupies less than 2K bytes of program space. This high level language approach coupled with selective manual optimization permitted far quicker and more error free development than could have been achieved using assembly language.

The approach to handling interrupts did introduce additional software overhead. For a typical configuration of the hardware, the realistic minimum interrupt latency would be about 200 μs. Using the message mechanism it is about 800 μs. For the targetted process control applications, this is entirely acceptable. RMX does make provision, however, for direct handling of selective interrupts which require better response time without disturbing the use of the message mechanism for the others. For normal task communication, the performance is relatively better. For the typical hardware configuration, the transmission of a message takes about 800 μs, which is comparable to the time that would be re-

quired for any synchronization primitive (e.g., *P* and *V* or enqueue and dequeue) on such hardware.

B. Engineering for Hardware Dependencies

The two functions which vary most significantly across the SBC product line are the timing and interrupt facilities. To accommodate these variations, the implementation separates the logical and physical parts of these functions.

The interrupt facilities are split between the module which implements the communications operations and a hardware interrupt handler module. The communications module provides a special "interrupt send" operation which performs the logical translation of the interrupt event into a message. This facility is independent of the interrupt structure of the processor board and remains the same in any version of RMX. The hardware dependent interrupt module deals directly with the hardware interrupt structure and invokes the send operation at the logical level. Only this module need be redesigned when generating an RMX version for a different SBC board. With this approach we take full advantage of the hardware vectored priority interrupt structure on high performance products and can simulate this desirable structure at slightly higher software cost on low performance products.

The same sorts of variations are faced in providing a source for the system time unit. Again, one module provides all of the logical time functions associated with providing time delays and time limits to the user system. This module is independent of the type, frequency, or location of the physical time source. A separate module is responsible for clocking the logical level by invoking it once every system time unit. Once again, this permits a consistent definition of time in RMX systems regardless of the sophistication of the available time source, and it limits the amount of reimplementation that is needed to support new SBC products.

C. Example Data Objects

As an example of the complex data objects defined in the system we will consider the task and exchange objects illustrated in Fig. 3. The task object is 20 bytes long and embodies the execution state and status of a task. It consists of pointers used to link it onto various lists of tasks in the system. These lists are used to queue a task at an exchange, link it to other ready tasks, or keep track of its maximum delay when waiting. It also contains the stack pointer of non-running tasks which is sufficient to supply the remaining task register values when the task next executes. Finally, the object contains the task priority, some status information describing the state of the task, and a pointer to auxiliary information about the task.

The exchange object is 10 bytes long and implements the mailbox concept described earlier, primarily by serving as the source of header information for lists of messages and tasks. Each of these singly linked lists is addressed with head and tail pointers located in the exchange object. All exchanges in the system are also linked together.

The exchange objects are operated upon by the SEND, WAIT, and ACCEPT instructions of the RMX abstract machine. These instructions generally alter the "value" or contents of these complex data objects. The task object is not the direct operand of any RMX instruction described above. Rather it is indirectly altered as a side effect of various instructions. Just as the user of floating-point objects on most machines needs to know the length and existence of instances of the object, but not its internal structure, so the internal structure of these objects is generally unimportant to the users.

TASK OBJECT

| DELAY LINK FORWARD |
| DELAY LINK BACKWARD |
| THREAD |
| DELAY MESSAGE |
| EXCHANGE ADDRESS |
| STACK POINTER |

| PRIORITY | STATUS |

| NAME POINTER |
| MARKER |
| TASK LINK |

EXCHANGE OBJECT

| MESSAGE HEAD |
| MESSAGE TAIL |
| TASK HEAD |
| TASK TAIL |
| EXCHANGE LINK |

2 BYTES
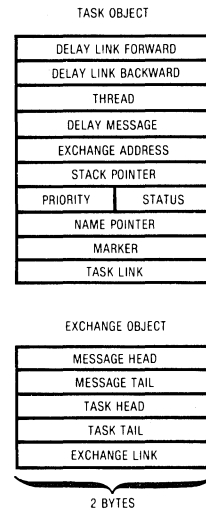
Fig. 3. Example data objects in RMX.

D. Global Versus Local Optimizations

We have already discussed some aspects of global versus local optimizations at the overall design level in terms of avoidance of redundant features. A good example of this tradeoff in the implementation is provided by the linked list data structures within RMX. Like many such systems there are a number of singly linked lists which must be maintained to reflect the status of the system. Local optimizations on the placement of links within data structures or in the form of the headers used for the lists would be guaranteed to save a few bytes of data space across the various lists. Further, the list insertion, scanning, and deletion algorithms could be specially tailored to the individual list structures to save microseconds of execution time for some operations on some lists. Indeed, any one such tailored algorithm might well use less code space than a single more general one.

On the other hand, many of the list operations are in no sense time critical. Generalizing all the list structures to use a single form replaces multiple algorithms with one, thus saving code space. The particular form can be chosen to favor those operations that are frequent, thus limiting the impact of the generalization on the execution speed of the system. Perhaps most important, however, is that, by reducing the number of algorithms and structures used, we decrease the potential number of errors and improve the maintainability of the resultant product. Since there are, for example, at least six distinct singly linked structures in the system, we reduce overall code size and engineering cost by supporting only a single mechanism. We improve product reliability at the price of a small increase in fixed data space and a small execution speed penalty of infrequent and nontime-critical operations.

It is interesting to note as an aside that this is really an example of software engineering: that is, applying engineering discipline to software development. Such discipline is highly valued and understood in other engineering fields. Standardized mechanical or electrical components are virtually always preferred to special designs; PLA's often replace random logic. Unfortunately, an appreciation of the overall benefits of such structure has been slow to develop in software engineering. Too often, we have seen special purpose designs and overly complex structure used in pro-

grams supposedly to save space or improve speed. The true costs in development time and reliability of such approaches have often been underestimated; the true time savings attributed to them often overestimated. The high percentage of end product cost due to software is finally forcing a general awareness of these issues.

VII. LSI AND ABSTRACT MACHINES

It seems natural at this point to ask how the abstract machine view of systems in general and our experience with RMX might be affected by the continuing development of LSI technology. Once we view any complex software system as defining a collection of abstract machines, it becomes obvious that it is simply an engineering decision as to which machines should be committed to hardware. We are constrained in this choice by the densities of our solid-state technology, the performance we desire, the applications that we are attacking, and perhaps most severely, by our understanding of software systems and of the machine structures that they require.

We might build an entire final application (e.g., a cash register) as a very-high-level single-chip machine. The specialization of such a design would, however, severely limit its application beyond the one for which it was specifically meant. On the other hand, we could build exclusively bit slice microprogrammable machines with utmost generality but which, due to their very low level of functional integration, would have no technological leverage for attacking complex problems. Actually, both these extremes have their well developed roles and will continue to be reasonable approaches for high-volume low-cost, and special-purpose tailored systems, respectively. It is in the middle ground —the area of the traditional computer—that directions are less clear.

If the 8080 type processors are generally somewhat less powerful than we actually need and as a result we always build operating systems of some level to support them, perhaps some of these functions can be integrated into the hardware. That is, if we can identify a broad range of systems which include essentially the same abstract machine implemented in software, then that abstract machine is a good candidate for hardware integration. The engineering difficulty is in understanding these software structures well enough to confidently and correctly commit them to hardware.

Attempting to build all of some very large and complex operating system onto one or two chips is, no doubt, out of the question with current technology. On the other hand, the final RMX system which we described resides in a small amount of ROM within the 65K address space of the 8080 processor. Once we view RMX as an abstract machine, the placement of the code which implements its functionality becomes immaterial. In particular, we could build an augmented 8080 type processor directly by defining the additional instruction codes of RMX as hardware operations and moving the RMX implementation into microcode on the chip. The resultant component would indeed be an "RMX machine" which dealt directly with the complex data objects and tables described above. It would have the advantage of not using any of the address space for operating system code. More importantly, it would not waste bus cycles and memory access time fetching operating system instructions. Such a machine would have the same advantages over a conventional one that a machine with floating-point hardware has over one without it.

Should we then try to build the RMX machine—ignoring for the moment whether our hardware technology is capable of it quite yet? Is the simple task model of RMX sufficiently general to be of use over a wide class of applications? Is the RMX machine the complete tool that we would like? Clearly, the answer is not a

wholehearted yes. For one example, RMX provides no isolation or protection of one task from another. Indeed, no solely software system can provide such protection at any reasonable cost. Such isolation would be desirable at the least because it would limit the damage that one task could do to another due to errors. The conclusion to be drawn, therefore, is not that this particular abstract machine should be built in hardware, but rather that some such machine would provide more of the facilities needed for building microprocessor applications than do current processors. Further, the design principles discussed above are the ones that appear most likely to be fruitful in creating such a machine.

VIII. CONCLUSIONS

In this paper, we have attempted to use a case study of a particular small operating system to illustrate both a philosophical approach to viewing computer systems and some important aspects of software development methodology. Many of the subtle aspects of desiging software to control quasi-parallel activities have not been discussed in detail, nor have we fully described the implementation. Nevertheless, we hope that this description suggests the practicality and necessity of disciplined approaches to software system design. Until software implementation reaches a level of engineering commensurate with that applied to other aspects of computer system design, our products will be very much bound by software costs. Only discipline and structure within our software efforts will ultimately permit microprocessor applications to reach their full potential.

REFERENCES

[1] P. Brinch Hansen, "The nucleus of a multiprogramming system," *Commun. ACM*, vol. 13, no. 4, pp. 238–241, Apr. 1970.
[2] ——, *Operating System Principles*. Englewood Cliffs, NJ: Prentice-Hall, 1973.
[3] F. P. Brooks, Jr., *The Mythical Man-Month*. Reading, MA: Addison-Wesley, 1975.
[4] W. L. Brown, "Modular programming in PL/M," in. *Proc. IEEE Conf. Computer Software and Applications*, Nov. 1977.
[5] K. Burgett and E. F. O'Neil, "An integral real-time executive for microprocessors," *Computer Design*, vol. 16, no. 7, pp. 77–82, July 1977.
[6] E. G. Coffman, Jr., and P. J. Denning, *Operating Systems Theory*. Englewood Cliffs, NJ: Prentice-Hall, 1973.
[7] G. W. Cox, "Portability and adaptability in operating system design," Ph.D. dissertation, Purdue Univ., Lafayette, IN, Dec. 1975.
[8] P. J. Denning, "Third generation computer systems," *Computing Surveys*, vol 3, no. 4, pp. 175–216, Dec. 1971.
[9] E. W. Dijkstra, "The structure of the 'THE'-multiprogramming system," *Commun. ACM*, vol. 11, no. 5, pp. 341–346, May 1968.
[10] J. H. Fasel, "Abstract machine hierarchies for programming language implementation,"Ph.D. dissertation, Purdue Univ., Lafayette, IN, Dec. 1977.
[11] A. N. Habermann, *Introduction to Operating System Design*. Chicago, IL: SRA, 1976.
[12] A. N. Habermann, L. Flon, and L. Cooprider, "Modularization and hierarchy in a family of operating systems," *Commun. ACM*, vol. 19, no. 5, pp. 266–272, May 1976.
[13] B. Liskov and S. Zilles, "Programming with abstract data types," *SIGPLAN Notices*, vol. 9, no. 4, pp. 50–59, Apr. 1974.
[14] D. D. McCracken, *A Guide to PL/M Programming for Microcomputer Applications*. New York: Wiley, 1977.
[15] D. Parnas, "A technique for software module specification," *Commun. ACM*, vol. 15, no. 5, pp. 330–336, May 1972.
[16] ——, "On the criteria to be used in decomposing systems into modules," *Commun. ACM*, vol. 15, no. 12, pp. 1053–1058, Dec. 1972.
[17] D. M. Ritchie and K. Thompson, "The UNIX time-sharing system," *Commun. ACM*, vol. 17, no. 7, pp. 365–375, July 1974.
[18] *RMX/80 System Users Guide*. Santa Clara, CA: Intel Corp., 1977.

3 iCS Products

iCS PRODUCTS

INTRODUCTION

In 1979, Intel introduced the Industrial Control Series — iCS product family. The iCS 80 chassis and the iCS 910, 920 and 930 Signal Conditioning/Termination Strips were the first members of the product family, followed closely by the introduction of the iSBC 569 Intelligent Digital Controller and the iSBC 941 Industrial Digital I/O Processor. The introduction of these products represents a new type of product offering from Intel — products designed specifically for the industrial control marketplace.

Publications reprinted in this section include an application note and article reprint on the iCS 80 chassis and termination strips, and an application note on the iSBC 569 Intelligent Digital Controller.

TABLE OF CONTENTS

Using Intel's Industrial Control
Series in Control Applications

Peter Andersen
OEM Microcomputer Systems Applications

9800932

Using Intel's Industrial Control Series In Control Applications

Contents

I. INTRODUCTION

The introduction of the single board computer as a tool for the system designer has opened the way for many varied application areas to benefit from the advantages of computer utilization. A problem still exists, however, because the available I/O configurations have been largely incompatible with the wiring and packaging techniques required in industrial environments. This problem is overcome by the utilization of the Intel® iCS™ product family. The purpose of this application note is to provide a representative approach to the implementation of a computerized solution to an industrial control system.

System Description

This application note will deal with a control system which will regulate the temperature in each of four ovens. Each oven will be defined as utilizing a light bulb for heating. Normal convection will be used to provide cooling. The internal temperature will be measured by means of a thermistor installed in each oven. We will assume that we will be required to implement some type of operator panel near the ovens which will allow the status of each oven to be monitored. This approach is similar to many common industrial applications which require a supervisory control station in one area and a separate operator interaction panel near the equipment being controlled. The setpoint and tolerances should be input from an external location.

With these facts about our system defined, we can begin a step by step solution to providing a computerized control system to operate the ovens. We will discuss the various equipment trade-offs and the decisions which will be used to define the hardware/software designs.

Control Algorithm

Before we can begin the design of our system, we must have a clear idea of the technique we will use to control the system. Our control system must maintain the oven temperature within a predefined and fairly narrow range of the setpoint. Let us make an assumption that the light bulb will be controlled digitally, meaning that the bulb must either be turned fully on or it must be turned fully off. The obvious control technique then becomes turning the bulb on when the temperature of the oven is below our lower limit and turning the bulb off when the temperature is above the higher limit. It seems reasonable to assume that this technique will provide a temperature in the oven which varies sinusoidally with time. This is true because even though the lamp is turned off, it will continue to generate heat for a short period of time. Likewise, when the bulb is turned on, it will not instantly be able to provide heat to raise the temperature of the
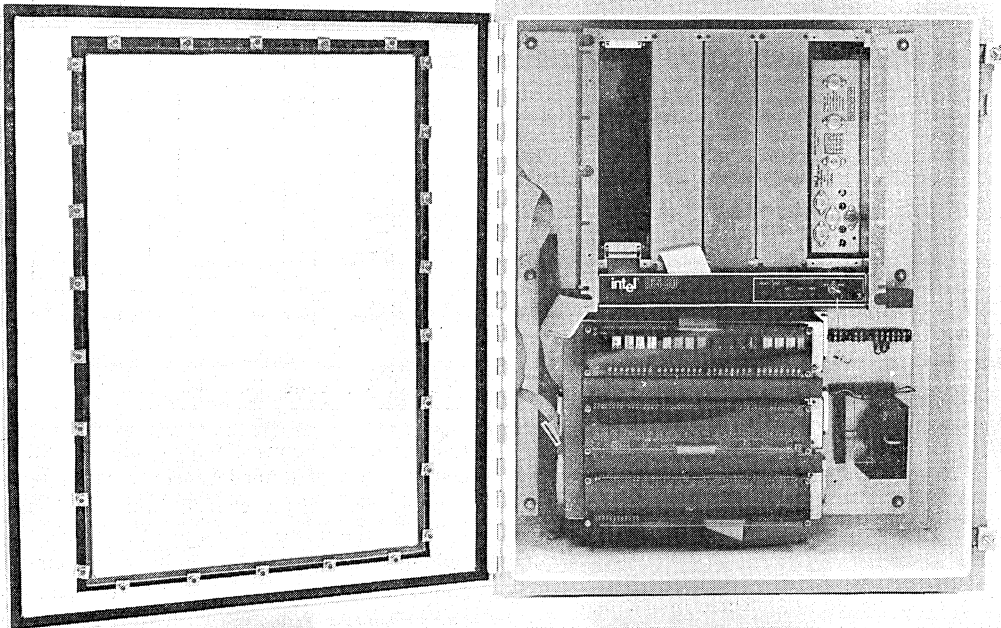
chamber. We would expect to have a system response such as is shown in Figure 1. A better method of control can be devised if we provide some type of temperature prediction into our control algorithm. Since this utilizes the rate of temperature increase or decrease, it will involve a type of derivative control system. This derivative control action will tend to dampen the temperature oscillations which might be encountered if only an instantaneous on-off control system were utilized. Figure 2 shows the response with time that we might expect with this type of control system.
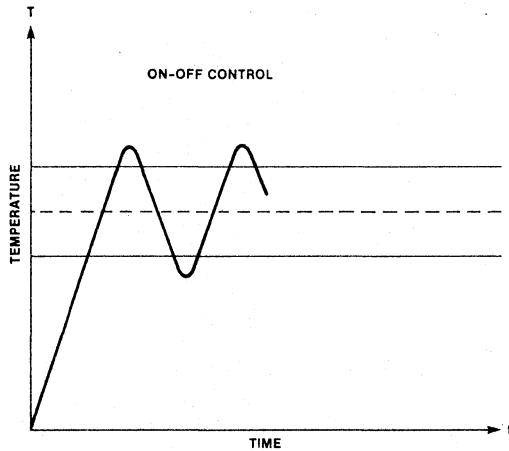
Figure 1. Maximum Effort Current Temperature
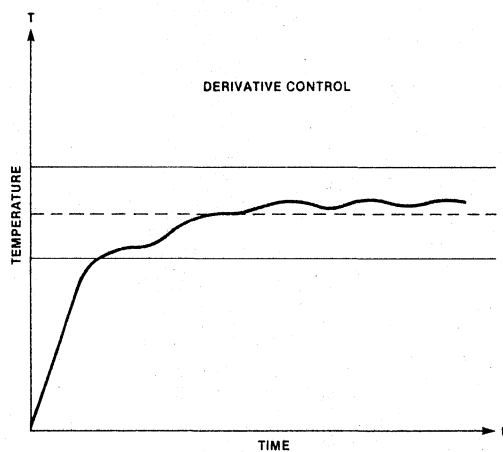
Figure 2. Maximum Effort Projected Temperature

The second approach is superior to the first because the control will provide a much smaller oscillation of the oven temperature. Other solutions are possible, such as providing a modulated output to the lamp. However, in an attempt to provide a simple model upon which to expand our system solution, we will assume that the second approach will provide us with an accurate enough control of the oven temperature.

Having made the decision as to the control technique, we can proceed with the task of determining the general system configuration. That is, we can define the physical system characteristics and the components to which we must interface the computer system. This approach is identical to that which would be used in a conventional control system design.

Basic System Configuration

Based upon the data which we have provided so far, it is possible to build a block diagram of the system's major components. The system consists of four ovens, an operator's panel, a data entry panel, and the actual control logic. A block diagram for the system is shown in Figure 3. We must now further define the elements which make up each of these blocks.
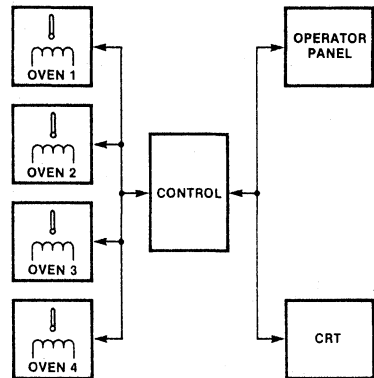
Figure 3. Application Block Diagram

Each oven must consist of a heating element, which we have already defined as being a light bulb, and a temperature sensing element which we have said will be a thermistor. Each heating element will be switched on or off by applying or removing a source of 115 VAC. The thermistor temperature can be sensed by using the thermistor in a voltage

divider circuit. We can then measure the voltage across a fixed resistor to obtain an analog signal which is proportional to the oven temperature. We will determine the required value of the fixed resistor at a later time.

The operator's panel should be designed to provide the workfloor operator with basic information as to the status of each oven. It should also allow some method by which he can inhibit the operation of any oven should it become necessary for charging or servicing the oven. We can then define the basic elements which should make up the operator's control panel. Each oven should have associated with it the following controls and indicators:

1. Oven ON/OFF Switch — This switch will allow the operator to inhibit the oven operation by turning the appropriate oven switch to OFF.

2. Oven RUNNING Indicator — This indicator will provide a visual indication that the oven is activated and that the temperature is being controlled.

3. Oven IN TOLERANCE Indicator — This indicator will turn on when the oven temperature falls within the allowable bandwidth around the setpoint for that oven.

4. Oven ALARM Indicator — This indicator is the complement of the in tolerance lamp. It will be turned on when the oven is activated and the temperature does not lie within the desired bandwidth.

5. Oven CAUTION Indicator — It may be necessary to alert the operator to a potential oven temperature control problem before it actually occurs and sets off the alarm indicator. Since we have defined our control algorithm as utilizing a type of derivative control, we can project the oven temperature ahead in time. We will turn the oven caution indicator on when we predict that the oven temperature will lie outside of the desired bandwidth in a predetermined future time period.

We have now defined the operator interface which we will utilize to control and monitor the oven processes.

At this point, we will make a decision that the interface used to input the setpoints will utilize a CRT terminal. Though the decision may seem to be completely arbitrary, we will see later that CRT terminals provide an extremely useful device for allowing an operator to communicate with the system. Once the decision has been made, we have no

further requirements to consider hardware design for this terminal, as the entire operation can be handled in the software development which will be considered later.

A common technique for documenting a system is the ladder diagram. At this time, we can construct a ladder for our control system. Unlike conventional design techniques, our ladder diagram need only be concerned with the actual drive and sensing circuits since the logic required to drive the various outputs will be defined using software. This results in a considerable simplification of the design process. A ladder diagram for a typical oven is shown in Figure 4. We can defer the implementation of the control algorithm until we begin to develop the software portion of our control system. It is now possible to complete the external hardware design and to implement the system wiring package.
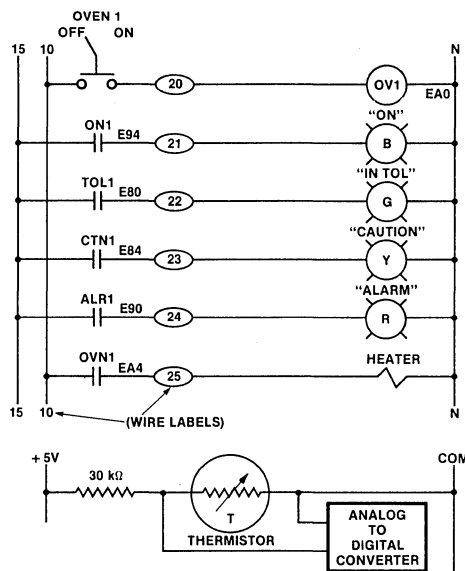
Figure 4. Ladder Diagram of One Oven

II. WIRING INTERFACES

A major pitfall in utilizing a computer for control systems has traditionally been the requirement for the design engineer to expend a considerable amount of his time in designing interfaces to connect the physical wiring to the computer system. The introduction of Intel's product line of termination panels has essentially eliminated the require-

ment of designing interfaces and allows more engineering time to be spent providing a solution to the application. Before we continue with the specific design, we should spend some time discussing the various types of termination panels available and the general characteristics of each panel.

Analog Termination Panels

The Intel® iCS 910™ Analog Termination Panel has been designed to provide a simple means of terminating the analog wiring and of providing an interface to the control system input/output. All wiring is terminated utilizing pressure type screw barrier blocks. Termination blocks have been provided to allow the termination of up to 32 single-ended or 16 differential channels of analog input. For use in a differential input environment, such as we will be using, the terminator blocks provide wiring terminations compatible with shielded cable inputs in that provision has been made to accept the shield of each input signal. The shield is then carried through the on-board circuits to the analog-to-digital converter. Provision has been made on the board for the mounting of commonly used circuits for signal conditioning. The available signal condi-

tioning circuits provide for installation of current termination resistors and the installation of a single pole low pass filter network. The basic barrier assignments for the iCS 910 termination panel are shown in Figure 5. The possible circuit networks for this panel are illustrated in Figure 6. A complete description of the analog termination panel can be found in the *iCS 910 Analog Signal Conditioning/Termination Panel Hardware Reference Manual* (manual order number 9800800A).

The functions of the analog termination panel will become more clear as we develop the actual configuration required to support our oven application. Referring to the ladder diagram (Figure 4) we see that a fixed resistor is necessary to provide the voltage divider network to sense the oven temperature. The current termination resistor (Rc) on the iCS 910 board can be used to provide a convenient mounting location for this component (refer to iCS 910 circuit schematic, Figure 6). At this point, we must make a design decision regarding the utilization of a low pass filter for our analog circuits. Since the oven temperatures are not expected to exhibit rapid fluctuations with time, the use of a low pass filter will not adversely effect the temperature
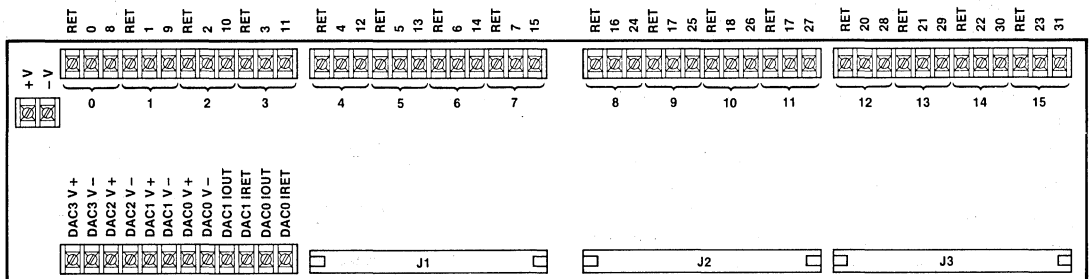
Figure 5. iCS 910™ Analog Terminator Panel Assignments
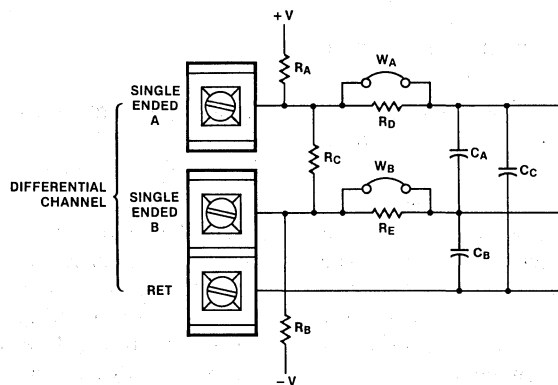
Figure 6. Typical Circuit on Analog Terminator

sensing. Indeed, the use of a low pass filter should contribute to spurious signal rejection should the analog cables pick up external noise signals. Calculations will show that the use of a filter network consisting of 11K ohm series resistors and a 2.2 μF capacitor will provide the filter characteristics shown in Figure 7.
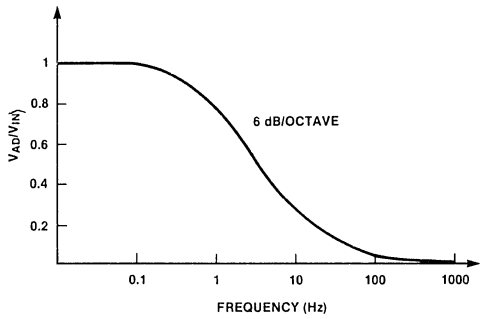
Figure 7. Single Pole Filter Characteristics

Based upon our requirements and using the circuit schematic of Figure 6, we can provide the circuit interfaces required by our ladder diagram (Figure 4) by configuring the channels of the iCS 910 terminator as shown in Figure 8. This results in a simple two-wire per oven analog interface. The terminator board is designed to connect to the various analog I/O boards by means of a standard ribbon

cable which is supplied with the terminator panel. The actual selection of the appropriate analog board will be deferred until later. We will define that oven number 1 will correspond to the differential analog channel 0; oven 2 will correspond with channel 1; oven 3 will correspond with channel 2; and oven 4 will use channel 3. This leaves 12 analog differential channels available for future expansion. The channel selection just made was a purely arbitrary choice.
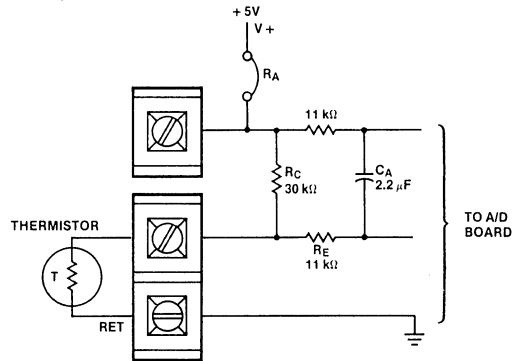
Figure 8. Analog Circuit for Oven Application

The wiring to the iCS 910 terminator panel can then be made essentially as shown in Figure 9. Clearly, the use of the terminator panel greatly simplified the connection between the control sys-
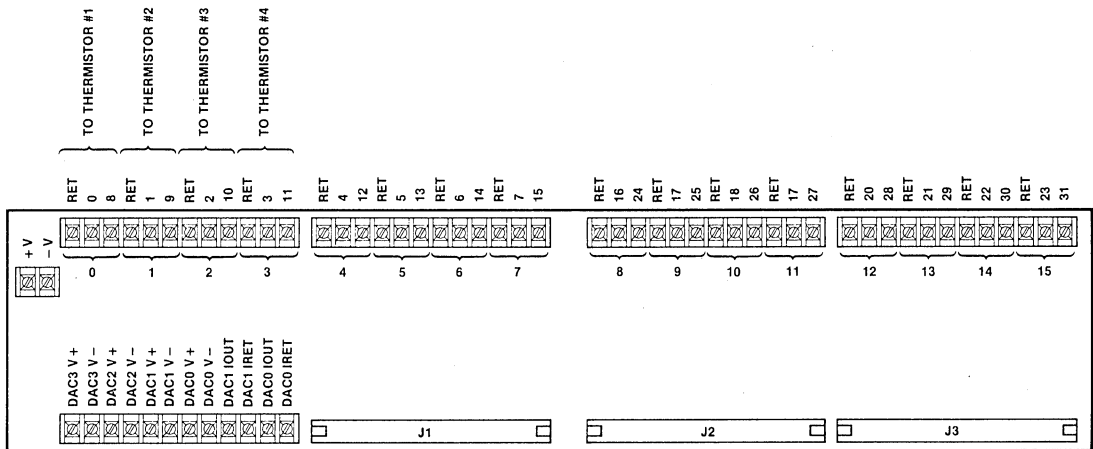
Figure 9. Analog Terminator Wiring

tem and the physical devices which are to be monitored or controlled. Figure 10 shows the placement of the components onto the board.
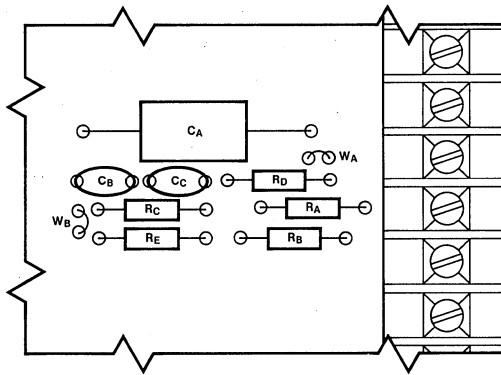
Figure 10. Analog Terminator Component Locations

Low Voltage Digital Termination Panels

Looking again at our ladder diagram for an oven control system (Figure 4), we see the need to provide a second type of interface signal. This is to provide the switching for the various indicator lamps used on the operator's control panel. Traditionally, this interface has been handled by using electromechanical relays. The coils would be driven by the low voltage control system and the relay contacts were used to drive the external indicators. Modern technology provides us with a solid state device to perform the same function, the optical isolator. We can use these devices to provide a highly reliably and low cost alternative to the relay interface. The Intel® iCS 920™ Digital Signal Conditioning/Terminator Panel provides us with a convenient vehicle for mounting the optical isolator circuits and for terminating the wiring associated with the indicator devices.

The iCS 920 panel is designed to be used by those interface circuits which incorporate operating voltages less than 50 volts and which generally use currents which are smaller than 300 mA. These limits are given only for a general guideline since a wide variety of optical isolators and drivers are available for use on the board. Some of the devices are capable of handling greater voltages or currents. A representative list of available devices and complete details of the termination panel are available in the *iCS 920 Digital Signal Conditioning/Termination Panel Hardware Reference Manual* (manual order number 9800801A).

The digital panel provides terminations for up to 24 digital channels, each of which can be configured as either an input or an output channel according to the specific application requirements. As with the analog termination panel, all wire terminations are made using pressure type barrier strips which will accept up to 16 gauge wire. The 24 digital channels correspond with those input/output channels assigned to the standard Intel I/O configurations used on the single board computers and I/O expansion boards. We will dwell more on this subject later when we define the addresses associated with each circuit which we desire to incorporate into the termination panel.

Since the digital channels can be configured into either an input or an output mode, it is wise to discuss each configuration so that a clear understanding of the board can be obtained, even though our application example will only use the output mode with this board.

Figure 11 provides a schematic of the panel when it is configured for a digital input mode. To set up a channel to operate as an input, it is necessary to add at least two jumpers to the wire-wrap jumper posts. As can be seen, pins 6 and 4 must be connected together as well as pins 3 and 5. If the board is to provide a visual LED indication of the channel status, an additional jumper should be installed between pins 1 and 2 of the jumper posts. If this is done, be certain to take into account the additional current requirements when calculating the required input resistors. Two resistor mounting locations are provided to allow installation of selected components to handle the current limit through the optical isolator (Rx) and the threshold voltage for turn-on of the device (Ry). A complete and detailed procedure for selecting these resistors based upon the input voltages is provided in the iCS 920 hardware reference manual mentioned earlier. Provision has also been made on the termination panel for the installation of a diode (CR) to protect against reverse bias application.

The components have been placed on the board arranged in groups of two channels. This eases the task of finding various components or of locating the holes for installing the required components. This layout is illustrated in Figure 12. It is important to take note of the physical placement of the optical isolator chips in the 20-pin socket. This installation location must be followed rigorously when using a channel in an input mode. Also take note that provisions are provided for mounting two sizes of resistors in location (Rx). This will accom-

modate the power dissipation requirements which will be encountered in various application situations. Referring again to Figure 12, note that the upper half of the layout represents odd channels and the lower portion of the layout is used for even channel component mounting.
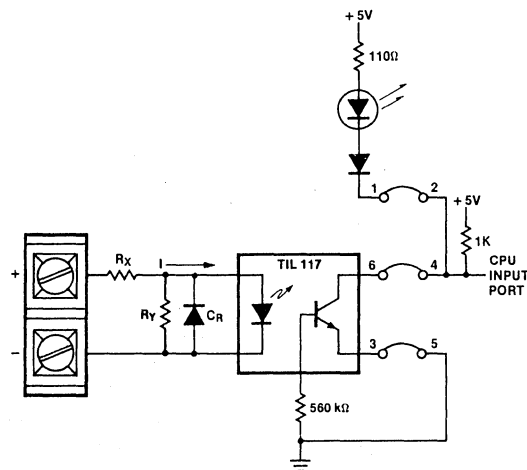
Figure 11. iCS 920™ Digital Terminator Input Configuration

When the iCS 920 panel is used in this input mode, it corresponds to the utilization of a relay coil to sense some external contact closure. The resistors can be thought of as selecting the coil's operating voltage and the diode provides the same transient

protection function as when installed on an electro-mechanical relay. Finally, the optical isolator output corresponds to the contacts associated with the relay coil. As we will see later, this approach provides us with an unlimited number of contacts per relay coil.

The oven application requires a contact for driving the indicator lamps associated with each oven. If we define the driving voltage to be 24 volts DC, we will find that the voltage and current requirements fall within the limits specified for using the iCS 920 Digital Signal Conditioning/Termination Panel. Let us examine in more detail how this can be accomplished.

We will select an industrial indicator assembly which utilizes a full voltage 24-volt lamp. Typical lamps would be type 387. This will require a drive of 40 mA at 28 volts. Our switching device must be capable of driving this load. The analogy used earlier to compare the optical isolator with a relay in an input mode holds true when we utilize the devices in an output configuration. If we examine the data sheet for the current switching characteristics of a typical optical isolator, say the TIL 113 (Appendix A), we can see that the current and voltage requirements fall well within the allowable ratings of the device. We have selected the relay contact characteristics! We need not concern ourselves with the selection of current limitation resistors (coil voltage ratings) since this circuitry is provided on the terminator panel when a circuit is
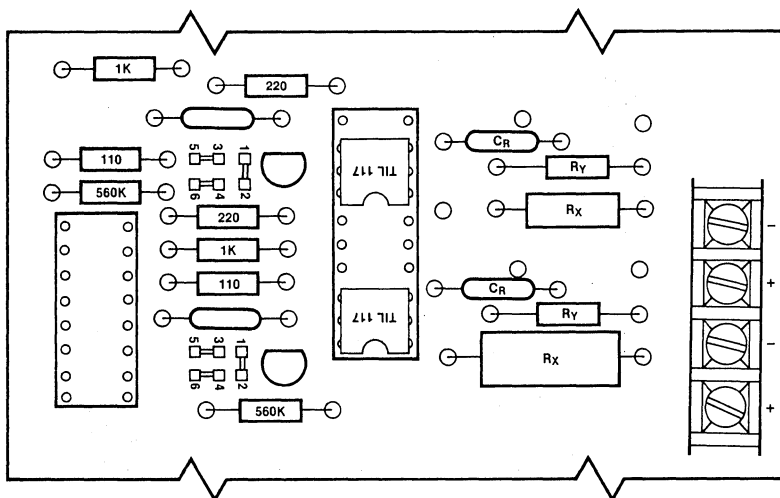
Figure 12. Digital Terminator Input Parts Layout

configured in an output mode. If we refer to Figure 13, we can see the on-board schematic for the output drive mode of operation. Two jumpers must be installed for each output channel. The first, between pins 1 and 2, is used to enable the LED channel status indicator. The second, between pins 3 and 4, actually connects the computer generated drive signal to the input of the optical isolator (analogous to connecting the relay coil to the driving line). Provision has been made on the circuit board for only one optional component in the output mode; this is the resistor (Rz). This component has the effect of increasing the response time of the switching device. Because our indicator lamps are not time critical, we will choose to omit the installation of this component.
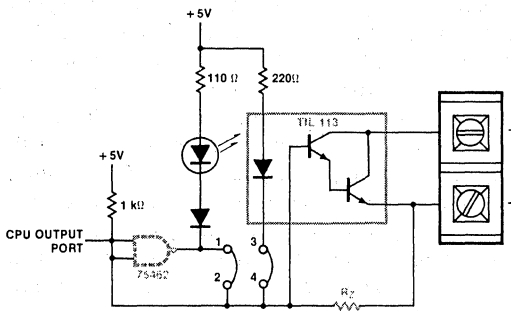
Figure 13. iCS 920™ Digital Terminator Output Circuit

Figure 14 provides a drawing showing the location of the components on the iCS 920 panel when it is utilized as an output switch. Again note the placement of the optical isolators in the 20-pin sockets. Also note the jumper arrangement used to provide the required output circuitry.

Again referring to Figure 13, we see that an alternative to using the optical isolator for a switch exists. Provision has been made on the panel for the installation of high power buffer/driver chips such as the TI 75462. This device provides the same coil/contact characteristics as our optical isolator; however, no isolation between the input and output is provided. In certain applications, this configuration may be desirable and can be implemented by connecting jumpers 1 and 3 together, then placing a jumper block in the isolator socket location. The oven application will not use this mode because of the many advantages which isolation can provide.

Prior to actually installing the components onto the iCS 920 panel, it is necessary to assign the lamps to definite channel addresses. This involves making some additional assumptions and design configuration decisions. If we consider the total number of digital inputs and outputs which are required to handle all four ovens (including the as yet unconsidered switch and heater signals), we see that a total of 24 channels will be required. These will be broken out as shown below:

No. of Channels	Type	Function
16	DC	Oven indicator lamps
4	AC	Oven heaters
4	AC	Oven RUN switches
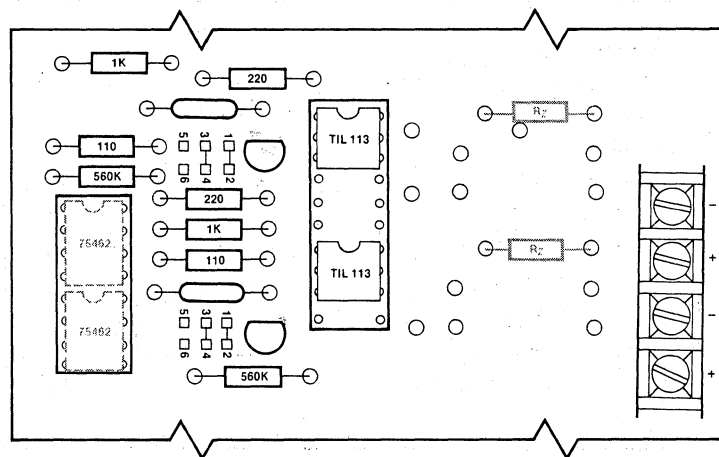
Figure 14. Digital Terminator Output Configuration

We have indicated that the 16 indicator lamps can be handled using the iCS 920 panel. An examination of the data sheets for the various Intel single board computers and expansion boards provides us with the fact that a common characteristic of most boards is the use of at least one Intel 8255 Programmable Peripheral Interface. This provides us with at least 24 I/O lines with which to work on each single board computer. We can then assume that we will not require an I/O expansion board to implement our application. Ideally, we can handle our total requirements with one parallel interface.

The various Intel parallel ports are brought off of the computer and expansion boards using edge connectors. These edge connectors are then connected to the termination panels using a standrd ribbon cable assembly, effectively providing an extension of the I/O ports out to the termination panels. The 24 channels are grouped into three I/O ports (each consisting of 8 channels or bits) which are then called port A, port B, and port C. When connected to the iCS 920 panel, these ports and their bit assignments will be as shown in Figure 15.

At this point, we seem to be in a dilemma since we would like to use all 24 channels and we have used only 16 of them on our panel while we have utilized the edge connector of the interface. It would be desirable to have some technique to extend the other 8 channels to a high voltage terminator panel. It might be well to interrupt our channel assignments at this time to jump ahead and consider the features of the iCS product line which will enable us to accomplish our interface desires. We will then consider the interface of the high voltage signals to our control system before returning to the problem of assigning port locations to our lines.

High Voltage Digital Termination Panels

The Intel® iCS 930™ AC Signal Conditioning/ Termination Panel is designed to interface up to 16 AC signals (up to 280 volts at 3 amps) or high current DC signals (up to 50 volts at 3 amps) to the parallel ports of the Intel single board computers or I/O expansion modules. The barrier strip terminations on this panel are designed to easily handle the 14 gauge wire commonly found in applications requiring the use of the AC terminator.

Solid state relays are used to provide the interface between the computer I/O ports and the physical plant devices. These devices make the utilization of the panel a simple task once a ladder diagram of the required circuits has been drawn. As we have previously mentioned and as is clear from looking at Figure 4, we shall need to utilize eight of the available circuits, four for input and four for output. The implementation of each signal type requires only that we insert the correct type of solid state relay into the appropriate socket.

First, consider the input configuration which is required to sense the position of the oven RUN switches. Figure 16 shows the circuit schematic when used in the input mode. We can see that the output signal will turn on when the input power is applied. Like the digital termination panel, each circuit's status is indicated by means of an LED indicator installed on the board. The input circuit is protected by a socketed 3-amp fuse which may be replaced without the need to solder any components. The solid state relay used for this configuration should be a type IAC5 which is available from either Opto-22 or Motorola. Complete details of available relays and their uses on the board are available in the *iCS 930 AC Signal Conditioning/*
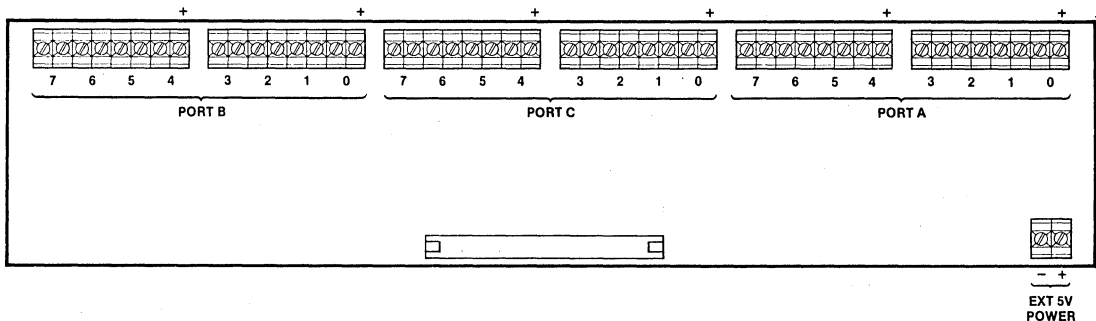
Figure 15. iCS 920™ Digital Terminator Port Assignment

Termination Panel Hardware Reference Manual (manual order number 9800802A). Keep in mind the fact that although this application note represents the solid state relays as being actual relays and contacts, they in fact are solid state and contain no moving parts.
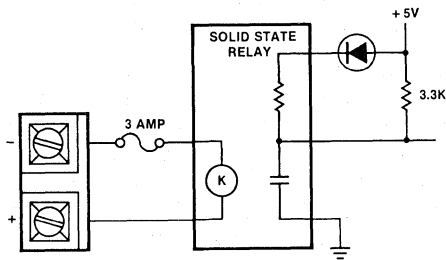
Figure 16. iCS 930™ AC Terminator Input Circuit

The output configuration is utilized to turn the heater elements (the light bulbs) on and off. Figure 17 provides us with a schematic of the output circuitry. In this case, we will insert a solid state relay of type OAC5 which will handle up to 140 volts RMS at 3 amps. In some cases, it might be desirable to add certain components to the terminator panel when using it in the output mode. Two possible circuit configurations are possible. The first and perhaps the most common will consist of installing a MOV (metal oxide varistor) across the solid state relay contacts. This will be required when the load being driven is inductive in order to prevent the transients generated by the load from damaging the triac in the SSR (solid state relay). Since the SSRs utilize zero voltage switching and the load in our ovens is resistive rather than inductive, our application will not necessitate the installation of this device. The second possibility for additional circuitry also involves driving inductive loads. When the load is highly inductive, a possibility exists that reliable operation of the SSR may not occur because of incorrect values for the dv/dt (a complete description of this phenomenon is available in various publications available from the manufacturers of the solid state relay devices). Provision has been made for installation of an external snubber network should this be required. Again, our oven control system will not require this type of circuitry. Figure 18 is provided for reference should the reader desire to see the location of the additional components on the panel. It should be noted that the component placement does not

allow the installation of the MOV and the snubber simultaneously.
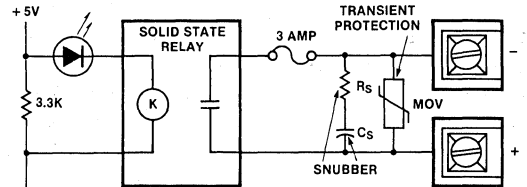
Figure 17. iCS 930™ AC Terminator Output Circuit
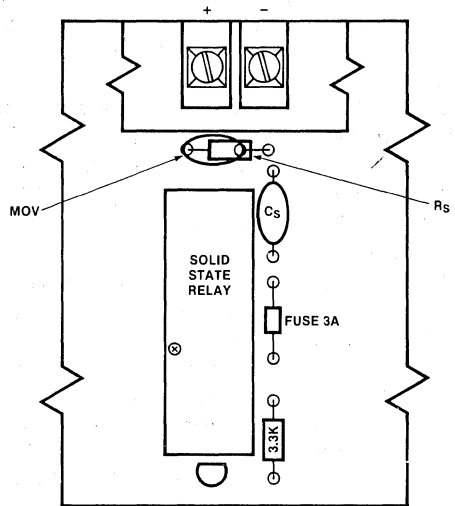
Figure 18. AC Terminator Component Locations

We can now get back to the task of assigning addresses to the various digital channels. The iCS 930 panel has three connector options for connecting it to the computer's I/O ports. The standard configuration utilizes connector J2 to attach the ribbon cable assembly. When this is done, the computer ports A and B will correspond to the 16 channels on the terminator panel (Figure 19). If we look at the termination panel, we will see that there is a provision for the user installation of two additional ribbon connector sockets onto the board. These are used in order to utilize the computer port C. If connector J3 is installed and utilized instead of J2, the channel assignments will be as shown in Figure 20. In a similar manner, connector J1 can be installed and utilized to provide connections between the computer port C and the other eight SSR positions. If we choose the 16 lines required for driving

the indicator lamps from the iCS 920 panel to be ports A and B, then it seems reasonable to assign the eight remaining lines required on the iCS 930 to port C. A feature of utilizing standard ribbon cable assemblies is the ability to easily add ribbon plug connectors to the cable. This will result in an assembly transferring ports A, B and C to the iCS 920 panel (however, port C is not used) and which continues the port C signals to the iCS 930 panel.

Individual channel assignments can now be made, grouping the inputs and outputs together in groups of four (this is done because of a requirement of the single board computers to share terminator and driver component packages in groups of four). Figure 21 provides a drawing showing the channel assignments and the physical wiring locations which will be used to connect the oven heaters and switches.
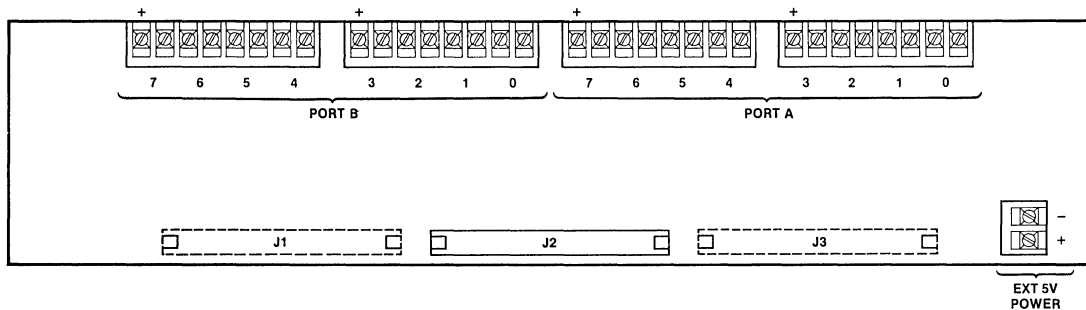
Figure 19. iCS 930™ AC Terminator Port Assignments
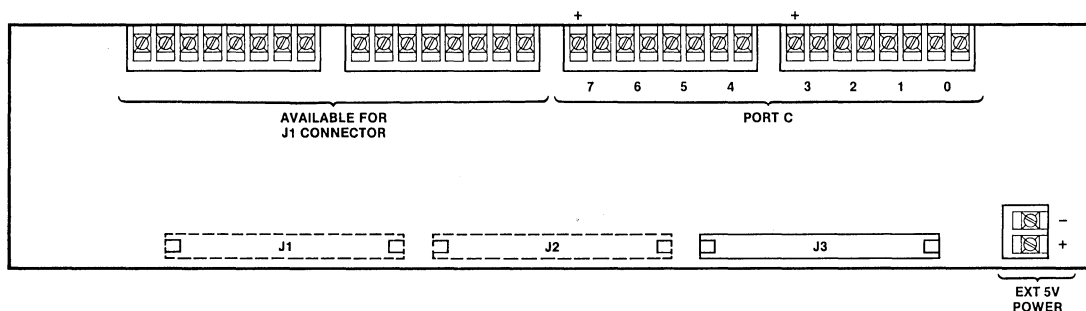
Figure 20. iCS 930™ AC Terminator Port Assignments
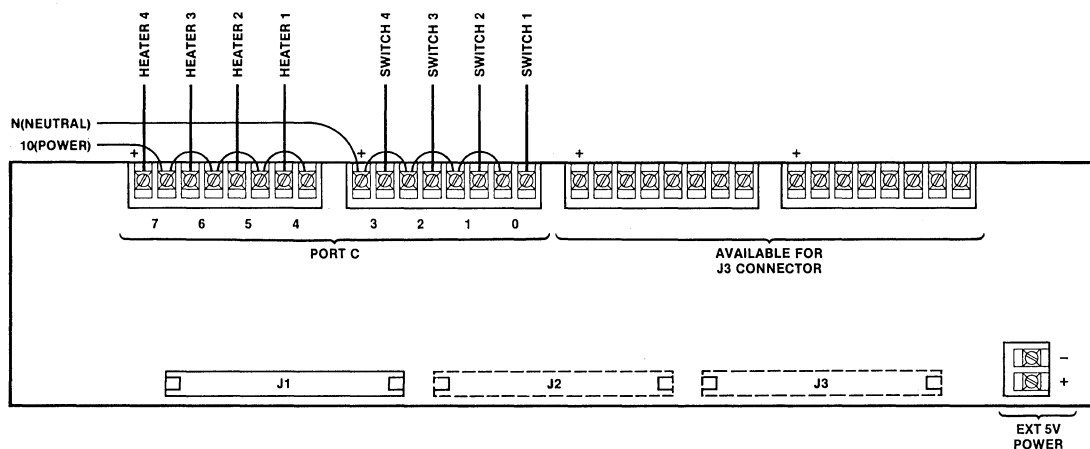
Figure 21. iCS 930™ AC Terminator Application Configuration

Final Channel Assignments

The only task remaining before we have completed our task of assigning channel numbers and physical wire and component locations is to assign these channels on the iCS 920 digital termination panel. Since we have already determined that we will utilize ports A and B, this becomes a simple matter, requiring only an arbitrary assignment of lamp locations using these port bits. The assignments made for one oven can be seen in Figure 22. The entire ladder diagram of the system can now be completed along with port assignments for all signals used. The completed diagram can be found in Appendix B. Note how the port assignments have been shown to the side of the ladder element representing that interface device.

The method used to define a port assignments needs to be clarified since it may not be apparent why a channel of port A was given the address of E80. To begin, we have already indicated that each port consisted of eight channels or bits. We will number these bits from 0 to 7. Since it is possible to have many input/output devices connected to the computer, the possibility exists of having multiple devices which incorporate internally ports A, B, and C. The computer has been designed to support up to 256 of these ports so we have numbered them using the hexidecimal numbering system. The possible port numbers can then range from 00 to FF. It will be found that a common characteristic of most single board computers is the use of assigning the port addresses of E8, E9, and EA to the on-board 8255 parallel peripheral interface. Therefore, the

first channel of port A would be defined as having an address of E80; the second channel of port B would be E91, and so forth.

III. SELECTING THE COMPUTER BOARDS

To this point we have delayed the selection of the boards which will be required to provide the computerized control system. The Intel OEM Microcomputer Systems Configuration Guide has been designed to simplify the task of selecting the required system. Our first task is to enter all known information describing our desired system into the project configuration worksheets. These worksheets can then be used to actually select a board configuration which meets our particular requirements. The effort required to accomplish the entry of data is reduced to a minimum through the use of predefined digital and analog configuration worksheets. Our requirement of having a total of 24 parallel data lines, consisting of a mix of high and low level interfaces, can be met by the 24-bit AC/DC combination. Our assignments of requirements for the terminator panels can be made and is shown in Figure 23. It can clearly be seen from the worksheets, that our required interface with the computer digital data will consist of one 24-bit wide connector (had we not used port C assignments, the use of 16-bit wide connectors would have sufficed). This means that our selected single board computer or I/O expansion board must provide at least one edge connector having 24 I/O bits on it.
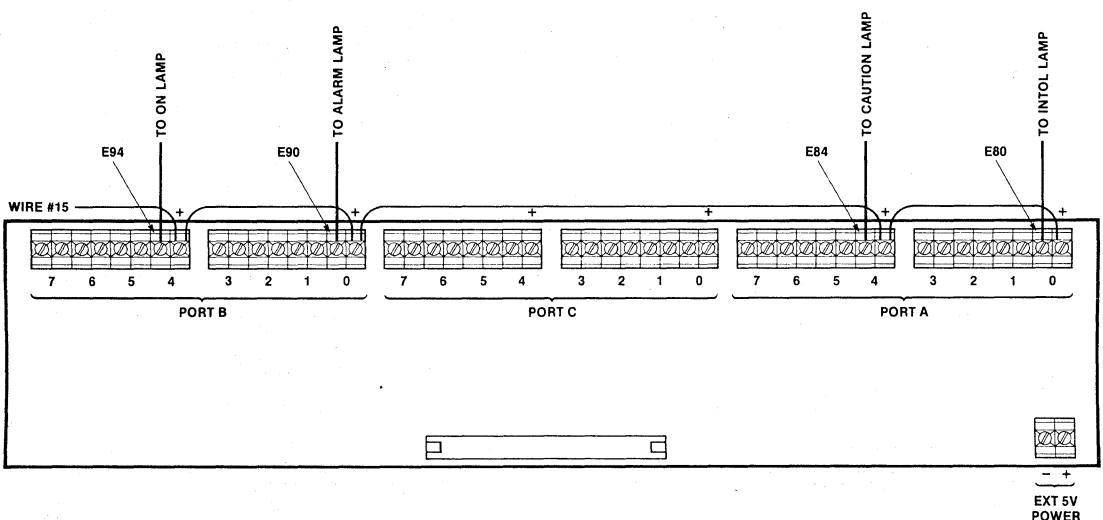
Figure 22. Digitsl Panel Application Configuration

DIGITAL CONFIGURATION WORKSHEET
PROJECT_____

This worksheet will provide the required digital interface configuration
data which is required to complete the Project Configuration Worksheet.

Enter Number of Channels

Enter # of Discrete AC Outputs (115-230 VAC) .... __4__ (A)
Enter # of Discrete AC Inputs (115-230 VAC) .... __4__ (B)
Enter # of Discrete DC Outputs (Current > 300 MA)............................. __O__ (C)
Enter # of Discrete DC Outputs (Current < 300 MA)............................. __16__ (D)
Enter # of Discrete DC Inputs .... __O__ (E)

Compute the Number of iCS 920™ and iCS 930™ Termination Panels

First compute the number of Parallel I/O ports (8-bits each port) required
on your iSBC™ board. Round all computations up to the nearest whole
integer unless instructed otherwise!

Compute # of iCS 930 Interface Output Ports ((A+C)/8) .... __1__ (F)
Compute # of iCS 930 Interface Input Ports (B/8).............................. __1__ (G)
Compute # of iCS 930 Termination Panels ((F+G)/2) .... __1__ (H)
Compute # of iCS 920 Interface Output Ports (D/8)............................ __2__ (J)
Compute # of iCS 920 Interface Input Ports (E/8) .... __O__ (K)
Compute # of iCS 920 Termination Panels ((J+K)/3) .... __1__ (L)

Optimization of Digital I/O Port Usage for Minimum I/O Configuration

Compute # of iCS 930 Output "Overflow Channels" DO NOT ROUND OFF)
(A+C)/8 .... QUOTIENT .... __O__ (M)
                    (Overflow Channels) REMAINDER .... __4__ (N)
Compute # of ICS 930 Input Overflow Channels (DO NOT ROUND OFF)
(B/8)..QUOTIENT .... __O__ (P)
                      REMAINDER .... __4__ (R)
Compute # of iCS 920 Output Overflow Channels (DO NOT ROUND OFF)
(D/8) .... QUOTIENT .... __2__ (S)
                      REMAINDER .... __O__ (T)
Compute # of iCS 920 Input Overflow Channels (DO NOT ROUND OFF)
(E/8).. QUOTIENT .... __O__ (V)
                      REMAINDER .... __O__ (W)
Compute 8-Bit Input Ports Required (P+V) .... __O__ (X)
Compute 8-Bit Output Ports Required (M+S).................................... __2__ (Y)
Compute 4-Bit Output Ports Required ((N+T)/4) (ROUND UP) .... __1__ (Z)
Compute 4-Bit Input Ports Required ((R+W)/4) (ROUND UP) .... __1__ (AA)
Compute 8-Bit Port C Requirements ((Z+AA)/2) (ROUND UP) .... __1__ (BB)
Total I/O Parallel Ports Required (X+Y+BB).................................... __3__ (CC)
Total # of 24 Channel Parallel I/O iSBC Board Edge Connectors
(CC/3) (ROUND UP TO INTEGER) .... __1__ (DD)

Compute Power Requirements for the Termination Boards
(DO NOT ROUND OFF)

Compute +5V for iCS 920 Board Outputs (.061×D) .... __.976__ (EE)
Compute +5V for iCS 920 Board Inputs (.023×E) .... __O__ (FF)
Compute +5V for iCS 930 Board Outputs ((.020×(A+C)) .... __.080__ (GG)
Compute +5V for iCS 930 Board Inputs (.012×B) .... __.048__ (HH)
Compute iCS 920 Power Requirements (EE+FF)................................. __.976__ (JJ)
Compute iCS 930 Power Requirements (GG+HH) .... __.123__ (KK)

Enter the appropriate data into the Project Configuration Worksheet as shown
below:

PROJECT CONFIGURATION WORKSHEET
EQUIPMENT PARAMETERS:

Figure 23. Digital Configuration Worksheet

The required power requirements of the termination panels can be calculated using the data provided in the digital configuration worksheet. The information regarding the necessary connectors and the power requirements should then be transferred to the project configuration worksheet (Figure 24).

Figure 24.

A similar technique is used to configure the analog signals using the standard analog configuration worksheet as shown in Figure 25. It can be seen that our application will require a single cable connection to a differential input edge connector of an analog input board. The power requirements can be calculated from the current requirements to drive the thermistors and the sensing resistors. The data is entered into the appropriate columns of the configuration tables and then transferred to the project configuration worksheet.

Figure 25.

The only remaining physical element of our control system which we have not defined is the CRT terminal which will be used for setpoint entry and modification. Communications with a terminal requires that we provide a serial RS232C port in our control system. This port requirement is entered

onto the worksheet and the system requirements are totaled as shown in Figure 26.

Figure 26.

We must now choose the Intel iSBC boards which will provide a solution to our system requirements. This is done by referencing the summary of key iSBC configuration parameters to find boards which provide the necessary characteristics. Our first task is to choose a single board computer which meets as many of our needs as is practical, while providing performance characteristics adequate to our needs.

Our first requirement for having support for a single RS232C serial communications channel can be seen to be met by a variety of possible boards. Among the possible boards meeting this requirement are:

iSBC 86/12™ iSBC 80/10A™
iSBC 80/20™ iSBC 80/20-4™
iSBC 80/30™

We must look further before a final choice can be made. Again, it can be seen that all candidates also meet the requirement of providing a minimum of one 24-bit wide digital I/O connector. Our decision must be based upon parameters which are not necessarily related to the input or output capabilities. Even though we have not yet developed our software package for our control system, we can safely make some assumptions regarding the completed software package and thus define additional requirements which will enable us to select our desired computer board. The software task will be considerably simplified if we write our programs in a high level language and if we use available drivers for our input and output where they are available. As we will see, the utilization of PL/M and RMX/80™ real-time executive and drivers will make this programming task much less demanding of our time. The trade-off is that these software tools take larger amounts of memory than if we were to write our entire application program in assembly language. Let us make an initial estimate that our system will require about 8K of EPROM and in the neighborhood of 2K of RAM.

Entering this data on the configuration worksheet (Figure 27) enables us to narrow our choice by eliminating the iSBC 80/10A since it does not have sufficient RAM on board.

Figure 27.

Since our application is not likely to require extensive math handling capabilities or high speed capabilities, we probably do not need the power found in the iSBC 86/12; so we will remove this product from consideration.

We are now faced with selecting either the iSBC 80/20 board or the 80/30 board for our processor. Each has certain advantages and disadvantages for use in our application. Let's compare these two boards, considering first the iSBC 80/20, then the iSBC 80/30.

> iSBC 80/20 board advantages — Slightly lower cost, greater number of I/O lines available.

> iSBC 80/30 board advantages — Faster processor, dual ported memory, able to utilize UPI modules.

If the system were to operate in a stand-alone environment and we could be certain that significant expansion would not take place, we would probably choose the iSBC 80/20 computer for our application. If we consider that the system might become a part of a much larger system by future expansions and additions, we should remember that the use of the UPI modules on the iSBC 80/30 computer provides considerable power through multiprocessing capabilities. The dual ported memory can also provide us with the ability to use more sophisticated inter-board communication protocol should the need arise. For the purposes of this application note, we will assume the system is being designed for expansion and we will select the iSBC 80/30 computer.

A good design practice is to provide an extra margin of available memory in the hardware design. Our anticipated RAM memory will use about 2K bytes. The computer will provide us with 4K bytes so we have a considerable margin. This is not true when we look at the amount of EPROM available on the board. Our 8K requirement is identical to

the amount of memory available to us on the board. We should consider the use of an expansion EPROM board or the prospect of having to spend a considerable amount of time reworking our program to get it to fit if we find that we have exceeded our estimates. We will select the option of adding a memory expansion board (it can be deleted if we find that our software requirements are less than estimated).

The computer selection and the memory expansion board data can now be entered onto the configuration worksheet as shown in Figure 28. If needed, the addition of the memory expansion board will allow our EPROM requirements to grow up to 16K bytes.

Figure 28.

The only requirement which we have not met is to assign a board to handle the analog input needs of our temperature sensing circuit. The analog voltage can be calculated and will be found to lie in the neighborhood of 4.6 volts at room temperature. This value will increase toward 5 volts as the temperature of the oven increases. Since we have no requirement for any analog output capabilities, we will choose the Intel® iSBC 711™ Analog Input Board to sense the voltage level. This board can be configured to handle a 5-volt full scale input and will provide a resolution of 12 bits. (If an oven requiring a wide range of temperatures and greater resolution were required, we would have to reconfigure our temperature sensor to provide a wider voltage spread over operating temperatures. For purposes of simplicity and clarity we will assume that our temperature resolution is adequate.)

The configuration worksheet can be filled in to reflect the selection of the analog converter and the total power requirements for the system can be computed as has been done in Figure 29. We now need to select a chassis and power supply in order to complete the application hardware design phase.

The Industrial Chassis

Before the boards can be operated together to form a control system, a means of allowing communica-

PROJECT CONFIGURATION WORKSHEET

EQUIPMENT PARAMETERS:

YOUR REQUIREMENTS	PRODUCT	MEMORY RAM (Bytes)	MEMORY EPROM (Bytes)	SERIAL I/O TTY	SERIAL I/O RS232C	PARALLEL IN	PARALLEL OUT	16	24	DI	SI	LOW	IO	VO	IN	OUT	+5V	+12V	-5V	-12V	List	Qty	Qty	Qty
iea iCS-910										4	0	0	0	0	1	0								
iea iCS-920						0	16										.976							
iea iCS-930						4	4	1									.125							
CRT				1																				
TOTAL		2K	8K	1		4	20	1	1	4					1		1.104							
SLOT #1 iSBC 80/30		16K	8K	1		4	20	1	1								3.500	.220	.0025	.050				
SLOT #2 iSBC 416			16K														.750							
SLOT #3 iSBC 711										8					1		1.700							
SLOT #4																								
SLOT #5																								
SLOT #6																								
SLOT #7																								
SLOT #8																								
SLOT #9																								
SLOT #10																								
SLOT #11																								
SLOT #12																								
ROM/EPROM																	.090							
I/O DRIVERS TERMINATION																	.420							
USER SUPP HARDWARE																								
SUBTOTAL																	8.668	.220	.0025	.050				

NOTE:
DI - Differential Input
SI - Single ended Input
IO - Current Output
VO - Voltage Output

SYSTEM INTEGRATION	SLOTS AVAILABLE	POWER AVAILABLE +5V	+12V	-5V	-12V
SYSTEM					
CHASSIS iCS 80-35	4	14.0	2.0	.900	.100
CARD CAGE					
POWER SUPPLY					
			TOTAL SYSTEM COST		

Figure 29.

tion between the boards and of distributing power among the boards must be found. This requirement is met by specifying a chassis into which the boards will be mounted. The Intel® iCS 80™ Industrial Chassis provides an environment for operating the boards which is specifically designed to operate in an industrial area.

The chassis has been designed to facilitate mounting into either a standard 19-inch RETMA cabinet or it may be rear-panel mounted into an enclosure such as may be found in applications requiring the use of a NEMA electrical enclosure. The card chassis has been mounted in such a manner as to hold the single board computers and expansion modules vertically, facilitating maximum cooling of the boards. Fans are provided to aid the normal convection cooling process. Card racks may be installed into the iCS 80 chassis to expand the card support capability to a maximum of 12 card slots in groups of four. Either an iSBC 635 or 640 power supply can be mounted into the industrial chassis to provide power up to 4 or 12 boards capability, respectively.

Our application design requires the installation of a three board solution, so we will choose the iCS 80 chassis with one iSBC 635™ power supply. We will choose to mount our control system in a standard NEMA 12 enclosure to protect the unit from the industrial environment. We should refer to the *iCS 80 Industrial System Site Planning and Installation Guide* (manual order number 9800798) for complete details for selecting appropriate enclosures and installation instructions.

The + 5 volt power needed to support the various termination panels and to supply a reference voltage for the thermistors is available from a barrier strip located on the lower front of the iCS 80 chassis (Figure 30). Our wiring can be routed to this barrier strip for those circuits requiring either 5-volt DC or the system logic common. A fuse holder is provided and a fuse should be installed for system protection. We will install a 2-amp fuse into the holder (our maximum power requirement for external circuitry should be 1.22 amps according to Figure 26).
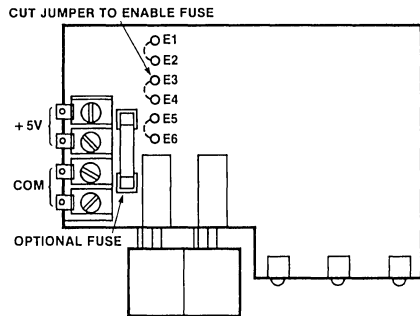
Figure 30. Industrial Chassis DC Power Strip

The remaining terms required in our ladder diagram (Appendix B) consist of a high voltage neutral and a source of switched high voltage power for the heater lamps. Both of these terms are available from the iCS 80 industrial chassis. It is desirable to utilize the same switched power for both the computer system and our external signals, so that we can provide protection to operators when one portion of the system is shut down. A common source will insure that all portions of the system are inactivated if repair is being done. The iCS 80 chassis incorporates a heavy duty industrial key-lock switch for its power switching. The outputs of this switch are available to the user at a terminal barrier strip located on a fold-out panel on the rear of the chassis assembly (refer to Figure 31). We can see that our neutral wire should be connected to terminal 5 (filtered AC low) and the wire for the AC high, wire #10 on the ladder diagram, should be connected to terminal 9. This will provide us with a switched, fused, and filtered power source for our external wiring.

As we will be installing the chassis into a NEMA enclosure, we will not want to use a standard power cord since this would involve the additional expense of installing a duplex outlet in the cabinet. The power wiring can be installed directly onto the power barrier strip by placing the AC hot wire on barrier number 1, the neutral wire onto barrier number 4, and the ground onto barrier number 3.

The hardware implementation of the system can now be considered to be complete. Before the system can function as a control for the oven temperatures, we must define the relationships between the various pieces of the oven system and we must also define the operator interface with the CRT terminal. Thus, we begin the software phase of our design.

IV. DETERMINATION OF SOFTWARE APPROACH

The task of providing the relationships between the various system components falls into the category of writing the software. Before we actually begin to develop this software, we will define certain guidelines which can be used to organize and simplify the task.

Let us consider the general environment under which our programs will operate. We find that we have essentially two choices in this area. First, we can consider the entire process as a sequential set of predefined operations in which we must perform each operation before moving to the next until finally we complete the sequence and begin again. (This is analogous to using a single stepper switch to design our control system.) Since each oven is independent of the others, we can not afford to use
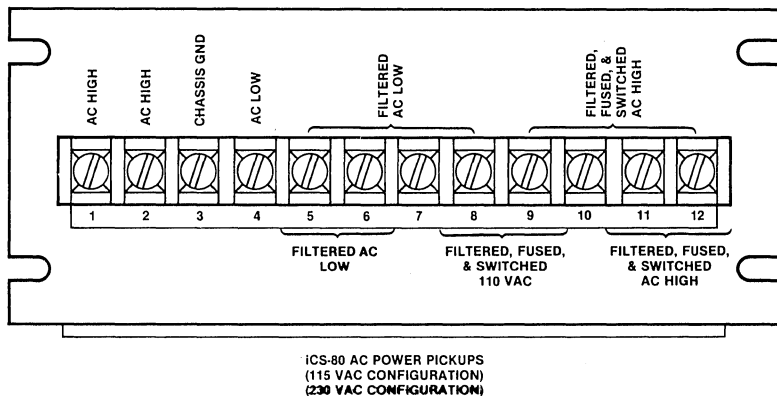
Figure 31. Industrial Chassis AC Power Strip

this approach since we could get tied up waiting for something to happen in a particular oven and would have to ignore the other ovens. The designer familiar with relay design will probably be thinking, at this point, that we should use a separate sequential operation for each oven or device to be controlled. Indeed, this is exactly what we can do with our software by using what is known as a real-time executive. This tool will allocate the computer's resources in such a manner as to provide us with the capability of having independent software programs or tasks operating at what appears to be the same time. We will make our first assumption that our software will be written using such a tool and we will specify that we will operate under Intel's RMX/80 Real-Time Multi-Tasking Executive. We will discuss more detail of this software tool as we develop our programs.

Next, we must consider the language which we will use to actually define our required operation. We have many alternatives from which to choose. Let us look at several of the alternatives in some detail.

Assembler

Assembler language is probably the most basic tool with which we can program a computer. It is considered to be the most efficient user of program memory and processor time. These features are made possible because each assembler instruction line is converted directly into a corresponding machine instruction. From a programming standpoint, assembler language is the most difficult to use since any task must be defined by subdividing that task into a multitude of smaller operations compatible with the available instructions of the computer. To use this language, we must be familiar with the architecture of each computer with which we desire to operate. The use of the language is somewhat simplified through the use of an Intel supplied assembler which converts the assembler code into machine instructions and provides listings of the operations which have been entered. A complete description of the Intel 8080/8085 Assembler Language is available in the *8080/8085 Assembly Language Programming Manual* (manual order number 9800301B).

The user should consider this programming tool when his application requires the minimum amount of memory (such as might be required for very large volume designs where memory cost is a factor) or where a highly time dependent routine

must be defined. Our oven application does not fall into either of these categories, so we will choose not to use this language in our instance.

PL/M

Intel's PL/M language offers an efficient, structured, high level systems programming language. Before proceeding, let us be clear on the benefits of using a high level language. First, the use of high level languages results in reduced development time and cost. High level languages provide the ability to program in a natural algorithmic language. In addition, they eliminate the need to manage register usage or to allocate memory. Second, high level languages provide improved product reliability because programs tend to be written in structured formats and result in a minimum of extraneous branches which might cause testing problems. Finally, their use produces programs which are better documented and are easier to maintain.

On the other hand, high level languages do not optimize the code segments as well as can be done by an experienced assembly language programmer. As a result, most compilers (routines which convert the high level languages into machine executable code) use more program storage than those written by the assembly language programmer. Different languages and compilers require different amounts of memory for the same task.

PL/M-80 is probably one of the most efficient high level languages for use on microcomputers. It has been determined that PL/M-80 users can expect to use between 1.1 to slightly more than 2 times as much program memory as would be used for the same task written in assembly language. For this reason, we must place the use of this language high upon our list of possible languages in this application.

A glance at the *PL/M-80 Programming Manual* (manual order number 98-268B) indicates that the language is highly structured and seems to lend itself very well to handle logical type operations. It seems to have the greatest weakness in its math handling capabilities in that it does not support negative numbers or fractions. It is reasonable to assume that the oven application can be handled entirely with positive integer numbers so this limitation will not unduly hamper our use of this language. We will keep these features in mind when making a final decision.

FORTRAN

Intel's FORTRAN-80 provides the full subset of ANSI FORTRAN 77. In many cases FORTRAN-80 has features that exceed the specifications for both the subset and the full versions of FORTRAN 77. Most of the power of this language lies in its ability to easily handle complex mathematical expressions. Obviously, it does not have any limitations regarding fractions or sign of the numbers involved. It should be used when the application requires the use of mathematical computations. The power of the language, however, means that the use of the language will take a heavy toll of memory allocation. A complete description of the FORTRAN version supported by Intel and its use on the iSBC computers can be found in the *FORTRAN-80 Programming Manual* (order number 9800481A) and in the *ISIS-II FORTRAN-80 Compiler Operator's Manual* (order number 9800480).

It is unlikely that the magnitude of mathematical routines required to control the temperature of our ovens will be complex enough to justify the use of FORTRAN. Keep in mind that, if such a situation were encountered, it is feasible to use a combination of programming languages to create our final module.

BASIC

Certaintly the most well known high level programming language today is BASIC. It offers a quick way of applying the computational capabilities of the computer to a wide range of applications. The Intel RMX/80 BASIC-80 is an interpreter designed to operate with Intel's single board computers and contains extended disk handling capabilities. As an interpreter, it differs from other high level languages in that it results in a relatively slower operating solution to an application. It is also not possible to use BASIC to generate multiple independent tasks which can compete for computer resources.

For these reasons, we cannot consider the use of BASIC for a solution to our application.

Final Selection of Language

From the above discussion, it seems clear that our choice for the application being demonstrated is to use PL/M-80 as our programming language.

With this in mind, we can begin the task of actually generating the code which will complte our application and provide an operating control system.

V. DEFINING SOFTWARE TASKS

The software implementation can begin as soon as we have broken our control functions into independent "tasks". We can then handle each task separately as though it were the only thing which had to be done by the control system. In the event that we find that one of our tasks must communicate with or be interlocked with another, we will handle this need through the use of "exchanges". The "exchange" can be thought of as a mailbox into which messages are deposited and picked up by the various tasks. These messages convey the necessary information between the otherwise independent programs. When all tasks have been coded, we will combine them using the facilities of RMX/80.

Our oven application can be broken down into three functional areas or tasks. These are:

1. The Control Task which will be used to actually sense the oven temperature and to provide the required responses to the heaters and the indicator lamps.

2. The CRT Update Task will be used to provide a "snapshot" of the system operations to a person viewing the CRT terminal.

3. The Parameter Update Task will be used to examine and update the oven setpoints and tolerances.

The choice of these three tasks has been essentially arbitrary in nature. Certainly, other choices and groupings of functions could easily have been made. We will use these choices for our example and will proceed with our development accordingly.

We have two other supporting tasks which must be included in our system. Fortunately, these tasks are predefined and fully supported within RMX/80's libraries; thus we need not write these functions. The two supporting tasks are:

4. A Terminal Handler Task to support the actual interface to the CRT terminal. It provides echo of input characters and signals when data is ready to be read. It will output messages to the terminal and signal when all characters requested have been sent.

5. An Analog I/O Driver Task to request and handle the handshaking which is required to communicate with the analog input board. It will signal us when data has been input and is available for use by our user written tasks.

We can proceed with the implementation of each of our three tasks which we have defined. The first step with each will be to develop a flowchart which shows the required operations to implement that task. This flowchart will show any intertask communications or exchanges that may be required with other tasks. The flowchart can then be coded using the facilities provided by our programming language.

Oven Control Task

The sequence of operations required to perform the control task can be defined using the flowchart shown in Figure 32. Let us examine the required steps in more detail.

An arbitrary decision has been made to only sample and control the ovens once each second. This will allow some time for the system to respond once a heater output has been set. The first step in our control task is to wait for one second to elapse.

Our next subtask should be to read the status of the various oven control switches on the operator's control panel. This item could wait until a later time, but there is no harm in handling it at this time.

Next, we see a block indicating the input of data regarding the current oven temperatures. This oven temperature data will certainly be used by the task handling the snapshot display on the CRT so we must give some consideration to the validity of the data. While we are in the process of getting the data and converting it to engineering units (next step), there will be periods during which the stored temperature data does not reflect the actual oven temperature. An example might be when we are actually moving the 16 bits of the temperature since we can only move data 8 bits at a time. During this period, we would not want another task to use the data and since each task is going to operate independent of others, we must provide some type of lockout of the data while we are operating on the temperatures (an alternative would be to have each task get its own temperature from the A/D converter and convert it to engineering units, but this would seem to waste memory and computer time). We can provide this lockout by creating an exchange to communicate with other tasks. If we make a message available in this exchange when the data is valid and cause no messages to be available when the data is nonvalid, we can effectively lock out tasks from using the data when it is in the process of being updated. This is done by requiring

those tasks to test for the presence of a message at the exchange before they get the temperature data. If no message is present, they must wait until one is placed into the exchange before proceeding. Just before we update the temperatures we will fetch the message from the exchange, leaving it empty while we work on the data. later we will again restore the message when the update is complete.
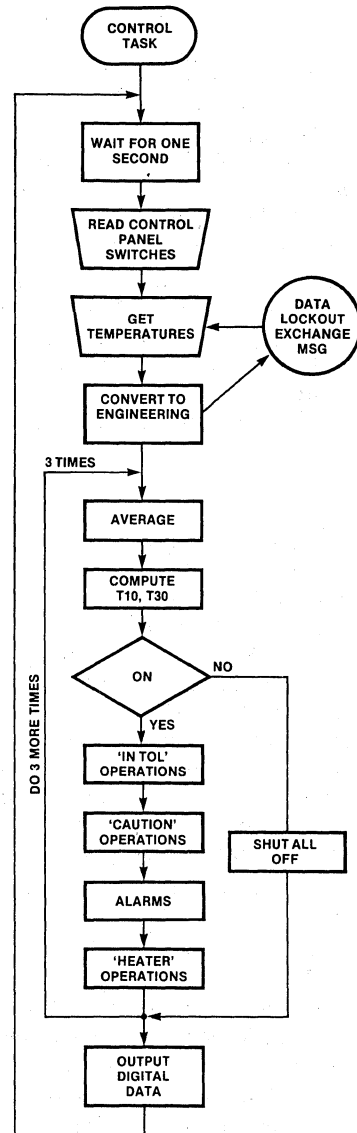
Figure 32. Control Task Flowchart

The number obtained from the analog converter provides us with a value which is proportional to the temperature of the oven. Our next step is to convert this number into engineering units. Unfortunately, the voltage and temperature are not related in a linear fashion since the thermistor is a nonlinear device. We will have to develop a technique to obtain a corrected value. For the purposes of this application note and in an attempt to keep the application as simple as possible, we have chosen to utilize a single table look-up to perform this conversion. Alternatives might have been to utilize FORTRAN routines to mathematically perform the conversion or to have separate tables for each oven. Once the conversion has been made, we must return a message to the data lockout exchange to allow other tasks access to the data.

Because we must deal with four ovens, the operations related to each individual oven must be performed four times, once for each. This is easily handled as we will see, since PL/M is a block structured language. Our flowchart need only remind us that the operations need be done four times.

The next step has been defined as performing some digital filtering of the temperature by averaging the current temperature with the temperature of one second ago. This filtered value will be used to perform subsequent computations and to make future decisions.

We have defined earlier in our definition of the control algorithm that we would use a derivative control. We have chosen to project the temperature ahead for a period of 10 and 30 seconds. We must calculate the rate of change and the temperatures in 10 and 30 seconds so that this data will be available when needed.

Now that the calculations have been made to determine numeric values required for the decision making process, we must begin the process of determining the status of each indicator and oven heater. A test will be made of the oven run switch and if it is found to be turned off, we will turn off all indicators and the oven heater associated with that oven. If the switch is found to be turned on, we will set the status of the "in tolerance", "caution", and "alarm" indicators according to our oven control algorithm. The oven heater will be turned on or off according to the projected temperature in 30 seconds.

Rather than output the individual oven indicator and heater data four times (once for each oven), we will perform the computations associated with making the decision four times (this saves code since we can use the same program steps with only pointers being exchanged). At the end of this time, a single operation will output the data to all ovens and indicators at the same time. Outputting to a computer port will actually cause the device to turn on or off according to whether the output bit is a one or zero.

We will then return to the beginning of our task to wait until another second elapses before we again perform the indicated functions.

Control Task Source Coding — The coding of our tasks is a straightforward procedure once we have prepared a flowchart. Since we are using PL/M-80 and RMX/80, the coding sequence for a task will be as follows:

1. Define any variables or structures which will be used in the module. This involves providing information defining variables as being either an 8 or 16-bit variable and declaring if that variable is to be a part of the task being coded or is to be found in some other task. If any arrays or structures are to be used, they must also be defined. Finally, if any program locations are to be used, they must be declared.

2. The task must be initialized. That is to say that any assumptions which will be made as to initial data values in subsequent instructions must be initially forced to this initial value.

3. The actual task must be coded to match the operations called out in the flowchart.

We will look at some examples of this coding process using the control task flowchart. The complete listing of this module and all modules actually used to provide the oven control system can be found in Appendix C.

At first glance, it would seem that the listing is extremely complex, but as we will see it is made up of straightforward pieces. The listing is made up of three parts as we have mentioned above when defining the steps required to generate a program. The first part (line numbers 1 through 50) is used to define parameters, variables, and external elements. The general types of elements making up this portion fall into typical categories. The first general category consists of DECLARE statements. Examples of typical lines will help explain their meanings (when actually developing the program, this first section was created piecemeal by

making an entry when it was found that a need for that term existed as the execution code in sections two and three were written).

Examples of the "declare" statement are shown below. For example, on line 11 we find:

11   1   Declare (n,k) byte;

This means that the variables "n" and "k" are being defined as terms which represent numbers or data which is one byte or 8 bits wide. The "11" is the program line number, and the "1" indicates that we are in the first level of nesting.

We can also see the use of the "literal" expressions such as used in line 4. The expression:

4   1   DECLARE FALSE LITERALLY '00H';

means that we are creating a new instruction called "false" and that its meaning is to be interpreted by the compiler as being equivalent to the value of zero.

Rather than dwell on the declaration, let us move on to the coding process which was used to generate the actual program. Keep in mind that the use of PL/M-80 requires that all terms used be declared in the program module. Refer to the *PL/M-80 Programming Manual* (order number 9800268B) for a full description of the PL/M language.

Program Initialization — The initialization portion of the program can be found on lines 51 through 59 of the control task program listing. This section is used to initialize data and to provide known entry conditions before we enter the repetitive program loop. This code is only executed when the system is reset or when the power is turned on. The control task requires two types of initializations; one to initialize the computer's output port and the other to set up the A/D converter. The requirements for each can be found in the RMX/80 User's Guide and the *iSBC 80/30 Single Board Computer Hardware Reference Manual* (order number 9800611A). Actual instruction examples are given in these manuals for the initialization operations.

Program Body — The program which actually provides the control operations can be found on lines 60 through 126 of the program listing for the control task. It has been divided into sections which correspond directly to the flowchart that was prepared earlier. Most instructions in PL/M-80 language follow closely the English structure which describes what is being done. The exceptions generally follow definite predefined formats. The for-

mat such as used on line 61 to wait for one second to elapse is an example of one such exception. Any time we desire to wait for a definite time period, we use an instruction of the form:

MSG$PTR = RQWAIT (.DUMMY$EXCH, TIME DELAY);

Whatever time delay we wish to use is expressed in increments of 50 msec time periods. Our example requires a time delay of one second so we will use the delay notation of 1.0/0.050 = 20 time units (this command is actually calling upon the RMX/80 executive to handle the delay).

The oven enable switch data has been defined by us to be routed by the hardware to the computer port "EA" which converts to a decimal number, 234. If we define an internal memory location for this data and call it BLOCK0, then we can get the oven switch data by using an input statement. Since the data sense is inverted through the hardware, we can provide meaningful internal data if the signal is re-inverted as it is loaded into memory. The instruction on line 62 of the control task listing performs this task.

We are now ready to get the analog data from the A/D converter. Our flowchart shows that we must lock out the other tasks from access to the temperature data during this time period, so we must first remove the enable message from the exchange in which it is stored. Messages are removed from an exchange by using an instruction of the form:

STORAGE = RQWAIT (EXCHANGE NAME,0)

Line 63 of the program listing means that we will get a message from our storage exchange which is called "Temp$lockout$exch" and store it in a memory storage area called "Lockout". Now, no other task can get a message from this exchange since it is empty, so it is permissible to operate on the temperature data. (Note how similar this command is to the one used to wait for a delay. Indeed, this is the same request for RMX/80, but it requests a time delay of zero.)

During the initialization, we built a message defining the characteristics of the analog signals and of the analog conversion board which we are using. Remember that we have indicated that the task of getting this data from the board is provided to us by one of RMX/80's predefined drivers. All that is necessary at this time is to inform that driver of our desire to get data, then wait until it has done its job and the data is available for us. The actual communication between our applications task and the analog driver is done using the idea of an exchange similar to that we have used to lockout the data.

We will send a message to the analog driver telling it what we want it to do, then we will wait until it sends a message back to one of our exchanges telling us that it is done. The format for sending a message to an exchange always follows the form:

CALL RQSEND (EXCHANGE NAME, MESSAGE NAME);

Line 64 of the listing shows that we have requested the input of the analog data since we have sent our message, Convert, to the analog driver's exchange which is called RQAIEX. We will wait until the operation is complete by using the line of code shown on the listing line 65. This is the same operation type that we used to get our message back providing a lockout earlier. The program will wait until a message is available before continuing.

The data must now be converted into engineering units. We earlier indicated that we would use a table lookup to perform the linearization, so we have included this table as a part of our program at line 50. The offset into the table corresponding to our temperature must be determined so that the correct value can be stored. Because we have four ovens, we will perform the operation four times with the data each time corresponding to the appropriate oven. These operations can be followed on lines 77 through 81 of the listing.

Lines 67 through 76 are used to establish an offset to be applied to the analog temperature data when the system is running. This program is only designed to be used during the start-up operations and is activated when a message containing a calibration request and current temperature is sent to its exchange.

The temperature lockout must be removed to enable other tasks to use this data. This is done on line 82 by sending the message back to the exchange used for intertask lockout communications.

The remainder of the program follows the flowchart and the operations can be followed using a flowchart and the listing. Each element of the flowchart corresponds to a block of code on the listing.
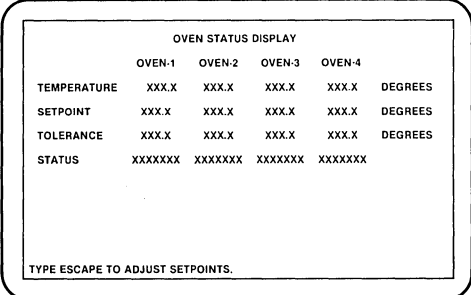
CRT Update Task Development

Earlier, we stated that the CRT update task would be used to allow the operator to view a "snapshot" of the four ovens. Let us turn our attention to developing the software which is required to accomplish this. We can begin by defining the elements which we feel should be displayed, then defining the format to actually be used with the CRT terminal.

Obviously, we need to provide the current temperature of each oven on our display screen. If we display the actual temperature, it seems reasonable to assume that we should also show the setpoint so that a determination can be made as to how well the system is performing. The control algorithm has been defined to use an allowable range to determine system outputs, so it would seen wise to also show this parameter. Finally, we should inform the viewer of the status of the oven so that he will realize that the reason an oven temperature is low is because the oven is off rather than an oven malfunction. Other items could be added if desired by the system designer, depending upon the total system requirements or the characteristics of the users.

We can now prepare a drawing of the CRT display to generate a layout of our desired characters and to generate an aesthetic display for viewing during operation. This drawing can be found in Figure 33.

Several techniques are available to output the required displays to the terminal. A decision must be made as to the frequency of screen updates; will we constantly refresh the data or do it only at certain intervals of time? If the terminal has the ability to disable the cursor, it makes sense to update data continuously. If the cursor cannot be disabled, its movement tends to be distracting, so the updates should be kept to a minimum. The terminal used for the application note did not have a disable feature, so we will make the decision to only update the screen once each second.

	OVEN STATUS DISPLAY				
	OVEN-1	OVEN-2	OVEN-3	OVEN-4	
TEMPERATURE	XXX.X	XXX.X	XXX.X	XXX.X	DEGREES
SETPOINT	XXX.X	XXX.X	XXX.X	XXX.X	DEGREES
TOLERANCE	XXX.X	XXX.X	XXX.X	XXX.X	DEGREES
STATUS	XXXXXXX	XXXXXXX	XXXXXXX	XXXXXXX	

TYPE ESCAPE TO ADJUST SETPOINTS.

Figure 33. CRT Status Display Layout

The decision to delay updates leads us to make another decision regarding the screen updates. If we only update a line which has data which has changed since the last update, the cursor movements will be kept at a minimum since it is unlikely that all parameters will ever change each second.

A flowchart can now be prepared showing the steps required to implement the CRT update task. This flowchart is shown in Figure 34. The coding of the program to support this task can be found in Appendix C. The development is identical with that which we described in the sections regarding the control task. Again, the software is divided into three parts, the declaration statements from lines 1 to 81, the initialization on lines 82 to 87, and the actual task code on lines 88 to 207.
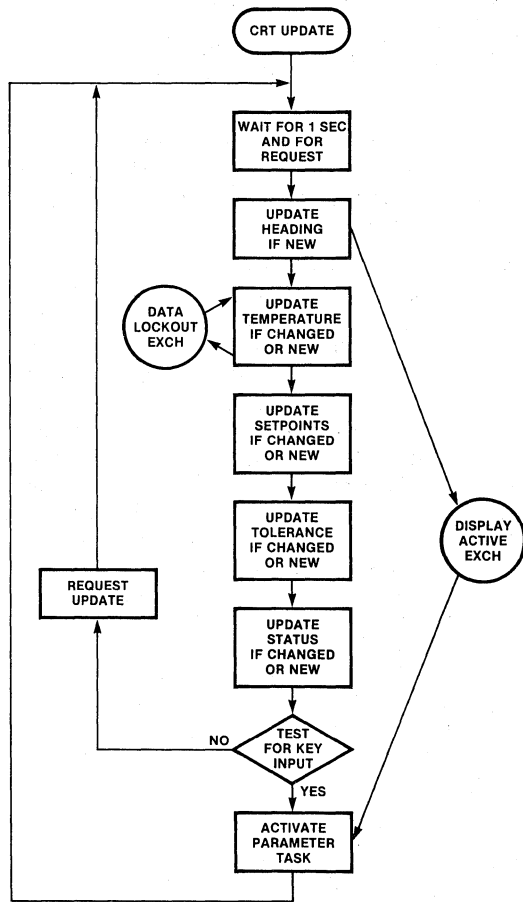
Figure 34. CRT Status Flowchart

A technique to exit from the CRT update mode and to get into a mode which will allow modification of the parameters has been introduced into the program and the display format. This is in the form of a message on the botton of the screen requesting the entry of an escape character to adjust setpoints. The software has been written in such a manner as to test for a character inut from the keyboard and if one is found corresponding to that character, the update task will allow the parameter update task to take control of the terminal (lines 190 to 204 of the listing).

Parameter Update Task

The parameter update task is used to actually allow the modification of the setpoints and the tolerances associated with each oven. A second use of the task is to provide a tool for establishing the zero offset associated with each analog channel so that an offset into the temperature linearization table can be computed by the control task.

Figure 35 shows the flowchart which describes the steps required to perform these operations. When the task has been completed, we will return to the CRT update task.
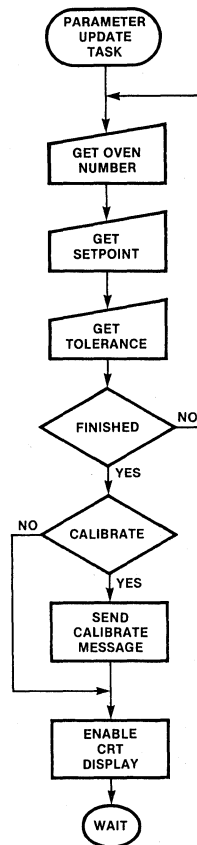
Figure 35. Parameter Update Flowchart

The program code for this task can be found in Appendix C and again follows the formats which we have discussed earlier. No attempt will be made in this document to provide a narrative of the listing since it follows the flowchart in development.

Support Programs

Three subprograms (procedures) have been written which provide functions which are common to the three tasks. This has been done to minimize repeating code segments thus saving as much memory as possible. These three subprograms support:

1. Conversion of a decimal string from the terminal into a binary number. This program is called ASC2BINARY and can be found in Appendix C.

2. Storage for common variables used by more than one task. These variables could easily have been included in other tasks but a purely arbitrary decision was made to include them in a separate module.

3. Conversion of binary numbers into a decimal string suitable for output to the terminal. This program is called DEC$REP and is found in Appendix C.

We now have completed the coding of the software to support our oven application. We must finish by combining all the software together to form a single loadable module.

VI. FINAL IMPLEMENTATION

When all code was linked and loaded to form an executable program module, it was found that the system required 9,041 bytes of EPROM and 1,735 bytes of RAM. These values fall within our hardware capabilities and will rquire that we program and insert nine EPROMs into the EPROM expansion card.

The system can now be tested and installed to control the ovens of our application. The actual system described in this application note has been constructed and tested. It has been found to control the oven temperatures of four ovens and performs as we anticipated when we developed our control strategy earlier in this application note.

VII. CONCLUSION

We have shown how Intel's single board computers, industrial chassis, termination panels, and software can be configured to provide a solution to a typical control application. We have seen how the development of a solution to a control problem can proceed along a predetermined and logical path. Truly, the utilization of the microprocessors can lead to optimum and cost effective solutions to control applications.

APPENDIX A
SELECTED DATA SHEETS

- Gallium Arsenide Diode Infrared Source Optically Coupled to a Silicon N-P-N Darlington-Connected Phototransistor

- High Direct-Current Transfer Ratio . . . 300% Minimum at 10 mA

- Base Lead Provided for Conventional Transistor Biasing

- High-Voltage Electrical Isolation . . . 1500-Volt Rating

- Plastic Dual-In-Line Package

- Typical Applications Include Remote Terminal Isolation, SCR and Triac Triggers, Mechanical Relays, and Pulse Transformers

mechanical data

The package consists of a gallium arsenide infrared-emitting diode and an n-p-n silicon darlington-connected phototransistor mounted on a 6-lead frame encapsulated within an electrically nonconductive plastic compound. The case will withstand soldering temperature with no deformation and device performance characteristics remain stable when operated in high humidity conditions. Unit weight is approximately 0.52 grams.

NOTES:
a. Leads are within 0.005 radius of true position (TP) at the gauge plane with maximum material condition and unit installed.
b. All dimensions are in inches unless otherwise noted.
c. Pin 1 identified by index dot.
d. Terminal connections:
1. Anode — Infrared-emitting diode
2. Cathode — Infrared-emitting diode
3. No internal connection
4. Emitter — Phototransistor
5. Collector — Phototransistor
6. Base (For TIL119, make no external connection) — Phototransistor

absolute maximum ratings at 25°C free-air temperature (unless otherwise noted)

Input-to-Output Voltage	±1.5 kV
Collector-Base Voltage (TIL113)	30 V
Collector-Emitter Voltage (See Note 1)	30 V
Emitter-Collector Voltage	7 V
Emitter-Base Voltage (TIL113)	7 V
Input-Diode Reverse Voltage	3 V
Input-Diode Continuous Forward Current at (or below) 25°C Free-Air Temperature (See Note 2)	100 mA
Continuous Power Dissipation at (or below) 25°C Free-Air Temperature:	
Infrared-Emitting Diode (See Note 3)	150 mW
Phototransistor (See Note 4)	150 mW
Total (Infrared-Emitting Diode plus Phototransistor, See Note 5)	250 mW
Storage Temperature Range	−55°C to 150°C
Lead Temperature 1/16 Inch from Case for 10 Seconds	260°C

NOTES: 1. This value applies when the base-emitter diode is open-circuited.
2. Derate linearly to 100°C free-air temperature at the rate of 1.33 mA/°C.
3. Derate linearly to 100°C free-air temperature at the rate of 2 mW/°C.
4. Derate linearly to 100°C free-air temperature at the rate of 2 mW/°C.
5. Derate linearly to 100°C free-air temperature at the rate of 3.33 mW/°C.

TEXAS INSTRUMENTS
INCORPORATED
POST OFFICE BOX 5012 • DALLAS, TEXAS 75222

electrical characteristics at 25°C free-air temperature

PARAMETER		TEST CONDITIONS[†]	TIL113 MIN	TIL113 TYP	TIL113 MAX	TIL119 MIN	TIL119 TYP	TIL119 MAX	UNIT
$V_{(BR)CBO}$	Collector-Base Breakdown Voltage	$I_C = 10\ \mu A$, $\quad I_E = 0$, $\quad I_F = 0$	30						V
$V_{(BR)CEO}$	Collector-Emitter Breakdown Voltage	$I_C = 1\ mA$, $\quad I_B = 0$, $\quad I_F = 0$	30			30			V
$V_{(BR)EBO}$	Emitter-Base Breakdown Voltage	$I_E = 10\ \mu A$, $\quad I_C = 0$, $\quad I_F = 0$	7						V
$V_{(BR)ECO}$	Emitter-Collector Breakdown Voltage	$I_E = 10\ \mu A$, $\quad I_F = 0$				7			V
$I_{C(on)}$	On-State Collector Current	$V_{CE} = 1\ V$, $\quad I_B = 0$, $\quad I_F = 10\ mA$	30	100					mA
		$V_{CE} = 2\ V$, $\quad I_F = 10\ mA$				30	160		
$I_{C(off)}$	Off-State Collector Current	$V_{CE} = 10\ V$, $\quad I_B = 0$, $\quad I_F = 0$			100			100	nA
h_{FE}	Transistor Static Forward Current Transfer Ratio	$V_{CE} = 1\ V$, $\quad I_C = 10\ mA$, $I_F = 0$		15,000					
V_F	Input Diode Static Forward Voltage	$I_F = 10\ mA$			1.5			1.5	V
$V_{CE(sat)}$	Collector-Emitter Saturation Voltage	$I_C = 125\ mA$, $\quad I_B = 0$, $\quad I_F = 50\ mA$			1				V
		$I_C = 10\ mA$, $\quad I_F = 10\ mA$						1	
r_{IO}	Input-to-Output Internal Resistance	$V_{in\text{-}out} = \pm 1.5\ kV$, See Note 6	10^{11}			10^{11}			Ω
C_{io}	Input-to-Output Capacitance	$V_{in\text{-}out} = 0$, $\quad f = 1\ MHz$, See Note 6		1	1.3		1	1.3	pF

NOTE 6: These parameters are measured between both input-diode leads shorted together and all the phototransistor leads shorted together.

[†]References to the base are not applicable to the TIL119.

switching characteristics at 25°C free-air temperature

PARAMETER		TEST CONDITIONS	TIL113 MIN	TIL113 TYP	TIL113 MAX	TIL119 MIN	TIL119 TYP	TIL119 MAX	UNIT
t_r	Rise Time	$V_{CC} = 15\ V$, $\quad I_{C(on)} = 125\ mA$,		50					μs
t_f	Fall Time	$R_L = 100\ \Omega$, $\quad$ See Figure 1		50					
t_r	Rise Time	$V_{CC} = 10\ V$, $\quad I_{C(on)} = 2.5\ mA$,					50		μs
t_f	Fall Time	$R_L = 100\ \Omega$, $\quad$ See Figure 1					50		

PARAMETER MEASUREMENT INFORMATION

Adjust amplitude of input pulse for:
$I_{C(on)} = 125\ mA$ (TIL113)
$I_{C(on)} = 2.5\ mA$ (TIL119).

47 Ω ── INPUT

V_{CC}

── OUTPUT
$R_L = 100\ \Omega$

TEST CIRCUIT

INPUT 0

OUTPUT

90% ... 90%

10% ... 10%

t_r ... t_f

VOLTAGE WAVEFORMS
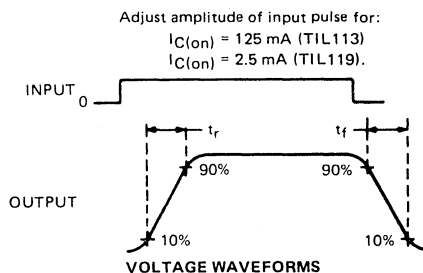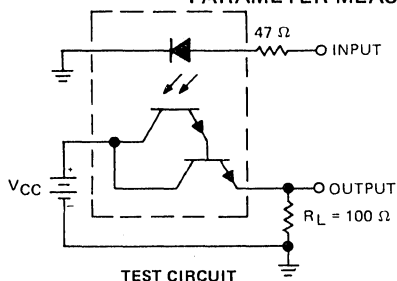
NOTES: a. The input waveform is supplied by a generator with the following characteristics: $Z_{out} = 50\ \Omega$, $t_r \leqslant 15\ ns$, duty cycle $\approx 1\%$, $t_w = 100\ \mu s$.
b. The output waveform is monitored on an oscilloscope with the following characteristics: $t_r \leqslant 12\ ns$, $R_{in} \geqslant 1\ M\Omega$, $C_{in} \leqslant 20\ pF$.

FIGURE 1—SWITCHING TIMES

TYPICAL CHARACTERISTICS

TIL113
COLLECTOR CURRENT
vs
COLLECTOR-EMITTER VOLTAGE

FIGURE 2

TIL119
COLLECTOR CURRENT
vs
COLLECTOR-EMITTER VOLTAGE

FIGURE 3

TIL113
COLLECTOR CURRENT
vs
INPUT-DIODE FORWARD CURRENT

FIGURE 4

OFF-STATE COLLECTOR CURRENT
vs
FREE-AIR TEMPERATURE
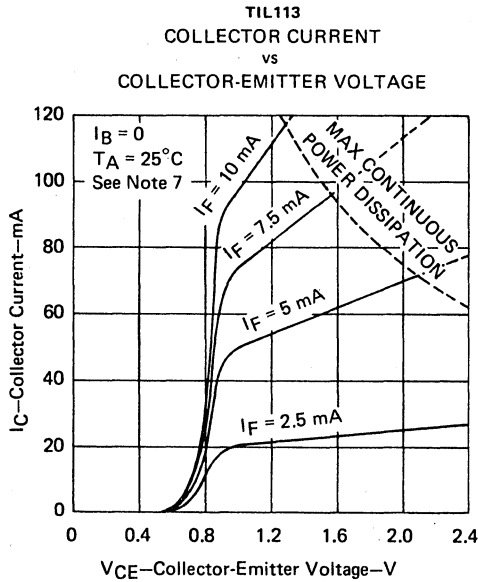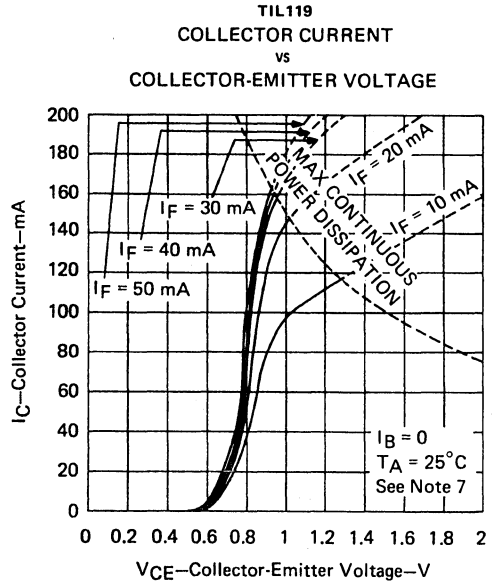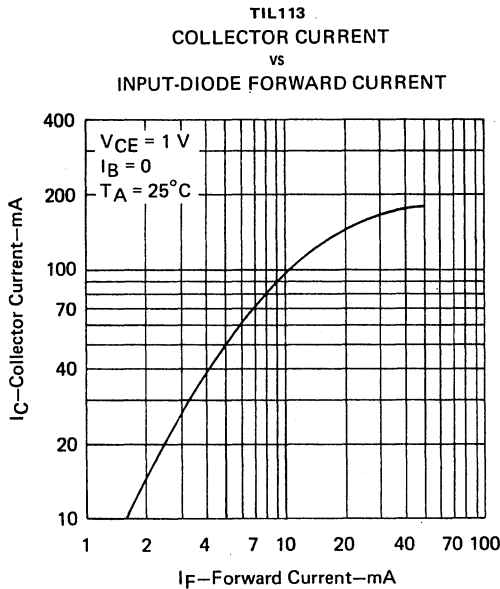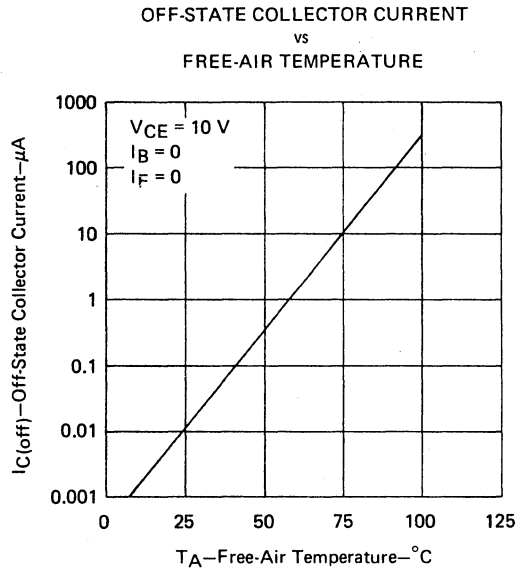
FIGURE 5

NOTE 7: Pulse operation of input diode is required for operation beyond limits shown by dotted line.

TYPICAL CHARACTERISTICS

TIL113
RELATIVE COLLECTOR-EMITTER
SATURATION VOLTAGE
vs
FREE-AIR TEMPERATURE

$I_C = 125$ mA
$I_B = 0$
$I_F = 50$ mA

$V_{CE(sat)}$ – Collector-Emitter Saturation Voltage Relative to Value at $T_A = 25°C$

T_A – Free-Air Temperature – °C

FIGURE 6

TIL113
TRANSISTOR STATIC FORWARD
CURRENT TRANSFER RATIO
vs
COLLECTOR CURRENT

$V_{CE} = 1$ V
$I_F = 0$
$T_A = 25°C$

h_{FE} – Static Forward Current Transfer Ratio

I_C – Collector Current – mA

FIGURE 7

INPUT DIODE FORWARD
CONDUCTION CHARACTERISTICS

See Note 8

$T_A = 25°C$

$T_A = 70°C$

$T_A = -55°C$

I_F – Forward Current – mA

V_F – Forward Voltage – V

FIGURE 8

NOTE 8: This parameter was measured using pulse techniques. $t_w = 1$ ms, duty cycle $\leqslant 2\%$.

TEXAS INSTRUMENTS
INCORPORATED
POST OFFICE BOX 5012 • DALLAS, TEXAS 75222

OPTO 22
I/O Module Detail Electrical Specifications

AC INPUT MODULES	MODEL IAC5	MODEL IAC15	MODEL IAC24	MODEL IAC5-A	MODEL IAC15-A	MODEL IAC24-A
AC INPUT LINE VOLTAGE	95 to 130 VAC →			180 to 280 VAC →		
INPUT CURRENT AT RATED LINE	10 ma →					
ISOLATION INPUT TO OUTPUT	2500 Volt RMS →					
INPUT ALLOWED FOR NO OUTPUT	1.5 ma →					
TURN ON TIME	20 Millisecond Maximum →					
TURN OFF TIME	20 Millisecond Maximum →					
OUTPUT TRANST. BREAKDOWN	30 Volts DC →					
OUTPUT CURRENT	25 ma →					
OUTPUT LEAKAGE 30VDC, NO. INPUT	100 Microamp Maximum →					
OUTPUT VOLTAGE DROP	.4 Volts at 25 ma Load →					
LOGIC SUPPLY VOLTAGE DC	4.5 to 6 V	12 to 18 V	20 to 30 V	4.5 to 6 V	12 to 18 V	20 to 30 V
LOGIC SUPPLY CURRENT	12 ma	15 ma	18 ma	12 ma	15 ma	18 ma

AC OUTPUT MODULES	MODEL OAC5	MODEL OAC15	MODEL OAC24	MODEL OAC5-A	MODEL OAC15-A	MODEL OAC24-A
LINE VOLTAGE	12 to 140 VAC →			24 to 280 VAC →		
CURRENT RATING	3 Amps[①] →					
1-CYCLE SURGE	55 Amps Peak →					
SIGNAL INPUT RESISTANCE	220 Ohm	1K Ohm	2.2K Ohm	220 Ohm	1K Ohm	2.2K Ohm
SIGNAL PICKUP VOLTS DC	3V 8V Ald.*	9V 16V Ald.*	18V 32V Ald.*	3V 8V Ald.*	9V 16V Ald.*	18V 32V Ald.*
SIGNAL DROPOUT VOLTS DC	1 Volt →					
PEAK REPETITIVE VOLTAGE	400V →		500 Volts →			
MAXIMUM CONTACT DROP	1.6V →					
OFF STATE LEAKAGE	5 ma RMS →					
MINIMUM LOAD CURRENT	20 ma →					
ISOLATION INPUT TO OUTPUT	2500 Volts RMS →					
CAPACITANCE INPUT TO OUTPUT	8 Pf →					
STATIC DV/DT	200 Volts/Microsecond Min →					
COMMUTATING DV/DT	Built in snubber (will commutate .5 power factor loads) →					

*Allowed

DC INPUT MODULES	MODEL IDC5	MODEL IDC15	MODEL IDC24
INPUT LINE VOLTAGE	10-32 VDC →		
INPUT CURRENT	32 ma at 32V →		
ISOLATION INPUT TO OUTPUT	2500 Volt RMS →		
CAPACITANCE INPUT TO OUTPUT	8 Pf →		
INPUT ALLOWED FOR NO OUTPUT	2 ma →		
TURN ON TIME	5 Millisecond Max →		
TURN OFF TIME	5 Millisecond Max →		
OUTPUT TRANST. BREAKDOWN	30 Volts DC →		
OUTPUT CURRENT	25 ma →		
OUTPUT LEAKAGE 30 VDC NO INPUT	100 Microamps Max →		
OUTPUT VOLTAGE DROP	.4 Volt at 25 ma →		
LOGIC SUPPLY VOLTAGE	4.5 to 6V	12 to 18V	20 to 30V
LOGIC SUPPLY CURRENT	12 ma	15 ma	18 ma

DC OUTPUT MODULES	MODEL ODC5	MODEL ODC15	MODEL ODC24
LOAD VOLTAGE RATING	60V DC →		
OUTPUT CURRENT RATING	3 Amps[①] →		
OFF STATE LEAKAGE	1 ma Max →		
ISOLATION INPUT TO OUTPUT	2500 V RMS →		
SIGNAL PICK UP VOLTAGE	3V 8V Ald.*	9V 18V Ald.*	18V 28V Ald.*
SIGNAL DROP OUT VOLTAGE	1Volt →		
SIGNAL INPUT RESISTANCE	220 Ohm	1K Ohm	2.2K Ohm
1 SECOND SURGE	5 Amps →		
TURN ON TIME	500 Microsecond →		
TURN OFF TIME	2.5 Millisecond →		

* Allowed
① Derate .033 Amps per degree C from 20° C

High Voltage DC Output Modules

DC OUTPUT MODULES	MODEL ODC5-A	MODEL ODC15-A	MODEL ODC24-A
LOAD VOLTAGE RATING	200V DC		
OUTPUT CURRENT RATING	1 Amps		
OFF STATE LEAKAGE	2 ma Max		
ISOLATION INPUT TO OUTPUT	2500 V RMS		
SIGNAL PICK UP VOLTAGE	3V 8V Ald*	9V 18V Ald*	18V 28V Ald*
SIGNAL DROP OUT VOLTAGE	1Volt		
SIGNAL INPUT RESISTANCE	220 Ohm	1K Ohm	2.2K Ohm
1 SECOND SURGE	5 Amps		
TURN ON TIME	500 Microsecond		
TURN OFF TIME	2.5 Millisecond		

*Allowed

Fast Switching DC Input Modules

DC INPUT MODULES	MODEL IDC5-B	MODEL IDC15-B	MODEL IDC24-B
INPUT LINE VOLTAGE	4-16 VDC		
INPUT CURRENT	14 ma at 5V		
ISOLATION INPUT TO OUTPUT	2500 Volt RMS		
CAPACITANCE INPUT TO OUTPUT	8 Pf		
INPUT ALLOWED FOR NO OUTPUT	1 Volt		
TURN ON TIME	50 Microsecond Max		
TURN OFF TIME	100 Microsecond Max		
OUT TRANSISTOR BREAKDOWN	30 Volts DC		
OUTPUT CURRENT	25 ma		
OUTPUT LEAKAGE 30 VDC NO INPUT	100 Microamps Max		
OUTPUT VOLTAGE DROP	.4 Volt at 25 ma		
LOGIC SUPPLY VOLTAGE	4.5 to 6V	12 to 18V	20 to 30V
LOGIC SUPPLY CURRENT	12 ma		

3-38

THIS DRAWING CONTAINS INFORMATION WHICH IS THE PROPRIETARY PROPERTY OF INTEL CORPORATION. THIS DRAWING IS RECEIVED IN CONFIDENCE AND ITS CONTENTS MAY NOT BE DISCLOSED WITHOUT THE PRIOR WRITTEN CONSENT OF INTEL CORPORATION.

115 VAC / 230 VAC

J1-N   J1-D
J2-b   J2-d
J3-b   J3-d

24 VDC Power Supply

OVEN 1 OFF ON
J3-C J2-C   J2-V J3-V   EA0

OVEN 2 OFF ON
J2-P J3-P   EA1

OVEN 3 OFF ON
J2-J J3-J   EA2

OVEN 4 OFF ON
J2-D J3-D   EA3

OVEN HEATER 1
EA4 J4-D J8-D   J8-E J4-E

OVEN HEATER 2
EA5 J5-D J9-D   J9-E J5-E

OVEN HEATER 3
EA6 J6-D J10-D   J10-E J6-E

OVEN HEATER 4
EA7 J7-D J11-D   J11-E J7-E

J3-B
J2-B

X10 +5 C 24

X10 +5 C 24   N

E94 J3-E J2-E   1 "ON"
E95 J3-K J2-K   2 "ON"
E96 J3-R J2-R   3 "ON"
E97 J3-W J2-W   4 "ON"
E80 J3-F J2-F   1 "IN TOL"
E81 J3-L J2-L   2 "IN TOL"
E82 J3-S J2-S   3 "IN TOL"
E83 J3-X J2-X   4 "IN TOL"
E84 J3-G J2-G   1 "CAUTION"
E85 J3-M J2-M   2 "CAUTION"
E86 J3-T J2-T   3 "CAUTION"
E87 J3-Y J2-Y   4 "CAUTION"
E90 J3-H J2-H   1 "ALARM"
E91 J3-N J2-N   2 "ALARM"
E92 J3-U J2-U   3 "ALARM"
E93 J3-Z J2-Z   4 "ALARM"

X10 +5 C 24   N

X10 +5 C 24   A/D CH. 0
30K   J4-A J8-A
J4-C J8-C
J4-B J8C

A/D CH. 1
30K   J5-A J9-A
J5-C J9-C
J5-B J9-B

A/D CH. 2
30K   J6-A J10-A
J6-C J10-C
J6-B J10-C

A/D CH. 3
30K   J7-A J11-A
J7-C J11-C
J7-B J11-C

X10 +5 C 24

REVISIONS
ZONE | REV | DESCRIPTION | DFT | CHK | DATE | APPROVED

DRAWN
ISSUED
SIZE C | CODE | DWG NO. | REV
SCALE | SHEET

9300020

USING INTEL'S INDUSTRIAL CONTROL SERIES IN CONTROL APPLICATIONS

```
                    $TITLE ('CONTROL TASK')
                    /*****************************************************
                    * This task handles the control and monitoring of    *
                    * four oven chambers.                                 *
                    *****************************************************/
1                   CONTROL$TASK$MODULE:
                    Do;
2       1           DECLARE EXCHANGE$DESCRIPTOR LITERALLY 'STRUCTURE (
                      MESSAGE$HEAD ADDRESS,
                      MESSAGE$TAIL ADDRESS,
                      TASK$HEAD ADDRESS,
                      TASK$TAIL ADDRESS,
                      EXCHANGE$LINK ADDRESS)';
3       1           DECLARE TRUE LITERALLY '0FFH';
4       1           DECLARE FALSE LITERALLY '00H';
5       1           DECLARE BOOLEAN LITERALLY 'BYTE';
6       1           DECLARE FOREVER LITERALLY 'WHILE 1';
7       1           DECLARE MSG$HDR LITERALLY '
                      LINK ADDRESS,
                      LENGTH ADDRESS,
                      TYPE BYTE,
                      HOME$EX ADDRESS,
                      RESP$EX ADDRESS';
8       1           DECLARE MSG$DESCRIPTOR LITERALLY 'STRUCTURE(
                      MSG$HDR,
                      REMAINDER(1) BYTE)';
                    /* AIMSG.ELT - ANALOG INPUT REQUEST MESSAGE FORMAT */
9       1           DECLARE AIMSG LITERALLY 'STRUCTURE(
                                MSG$HDR,
                                STATUS ADDRESS,
                                BASE$PTR ADDRESS,
                                CHANNEL$GAIN ADDRESS,
                                ARRAY$PTR ADDRESS,
                                COUNT ADDRESS,
                                ACTUAL$COUNT ADDRESS)';
                    /* AITYP.ELT - ANALOG INPUT MESSAGE TYPES */
10      1           DECLARE AIREP LITERALLY '30',
                            AISQS LITERALLY '31',
                            AISQV LITERALLY '32',
                            AITRAN LITERALLY '33';
11      1              Declare  (n,k) byte;
12      1           Declare (MSG$PTR,LOCKOUT) address;
13      1              Declare (BLOCK0,BLOCK1,BLOCK2,BLOCK3) byte external;
14      1           Declare TOLERANCE(4) address external;
15      1           Declare TEMP(4) address external;
16      1           Declare SETPOINT(4) address external;
17      1           Declare T$AVERAGE(4) address;
18      1           Declare T$LAST(4) address;
19      1           Declare T$LAST$AVERAGE(4) address;
20      1           Declare T$t5(4) address;
21      1           Declare T$t10(4) address;
22      1           Declare STATUS(4) byte external;
23      1              Declare CRT$DISPLAY$LOCK(5) address external;
```

```
24   1        Declare TEMP$CALIBRATE(5) address external;
25   1        Declare DUMMY$EXCH(5) address external;
26   1        Declare TEMP$LOCKOUT$EXCH(5) address external;
27   1        Declare RQAIEX(5) address external;
28   1        Declare A$D$EXCH(5) address external;
29   1        Declare CONSTANT$LOCKOUT$EXCH(5) address external;
30   1        Declare CRT$STATUS$EXCH(5) address external;
31   1          Declare ALARM$MSG structure (MSG$HDR);
32   1        Declare CONVERT ai$msg;
              /* This term is used to convey initial temperatures */
33   1        Declare CAL$TEMP based MSG$PTR structure (
                  MSG$HDR,
                  CAL address );
34   1          RQWAIT:
                  Procedure (EXCH,MESSAGE) address external;
35   2          Declare (EXCH,MESSAGE) address;
36   2        end RQWAIT;
37   1          RQSEND:
                  Procedure (EXCH,MESSAGE) external;
38   2          Declare (EXCH,MESSAGE) address;
39   2        end RQSEND;
40   1          RQACPT:
                  Procedure (EXCH) address external;
41   2          Declare EXCH address;
42   2        end RQACPT;
43   1          Declare OVEN$IN$TOL(4) byte data (
                  01H,02H,04H,08H );
44   1        Declare OVEN$CAUTION(4) byte data (
                  10H,20H,40H,80H );
45   1        Declare OVEN$DANGER(4) byte data (
                  01H,02H,04H,08H );
46   1        Declare OVEN$ON$MASK(4) byte data (
                  01H,02H,04H,08H );
47   1        Declare OVEN$HEATER(4) byte data (
                  10H,20H,40H,80H );
48   1        Declare OVEN$RUN(4) byte data (
                  10H,20H,40H,80H );
49   1        Declare OFFSET(4) address;
50   1        Declare TABLE(256) address data (
                  200,201,202,203,204,205,206,207,208,209,
                  209,210,211,212,213,214,215,216,217,218,
                  219,220,221,222,223,224,225,226,227,228,
                  229,230,231,232,233,235,236,237,238,239,
                  240,241,243,244,245,247,248,249,250,251,
                  252,254,256,257,258,259,260,261,263,265,
                  266,267,268,269,270,271,273,274,276,278,
                  279,280,282,284,285,287,288,289,290,291,
                  293,295,296,298,299,300,302,304,305,307,
                  308,309,310,312,314,315,318,320,322,324,
                  326,328,330,332,334,336,338,340,342,344,
                  346,348,350,352,354,356,358,360,362,364,
                  366,368,370,372,374,376,378,380,382,385,
                  388,390,392,395,398,400,402,405,407,410,
                  412,415,418,420,423,426,428,430,433,436,
                  439,441,444,447,451,454,457,460,463,466,
```

```
            470,473,476,480,484,488,492,496,500,504,
            507,511,515,519,523,527,531,535,540,545,
            550,555,560,565,570,575,580,585,590,595,
            600,605,610,615,620,625,630,635,640,645,
            650,655,660,665,670,675,680,685,690,695,
            700,705,710,715,720,725,730,735,740,745,
            750,000,000,000,000,000,000,000,000,000,
            000,000,000,000,000,000,000,000,000,000,
            000,000,000,000,000,000,000,000,000,000,
            000,000,000,000,000,000                        );
```

```
                        /* Initialization of control task */
51      1       CONTROL$TASK:
                Procedure public;
52      2           Output(235)=81H;
53      2           CONVERT.BASE$PTR=0F700H;
54      2           CONVERT.LENGTH=21;
55      2           CONVERT.TYPE=AISQS;
56      2           CONVERT.RESP$EX=.A$D$EXCH;
57      2           CONVERT.CHANNEL$GAIN=0;
58      2           CONVERT.ARRAY$PTR=.TEMP;
59      2           CONVERT.COUNT=4;
60      2           Do forever;
                    /* Wait for one second to elapse */
61      3               MSG$PTR=RQWAIT (.DUMMY$EXCH,20);
                    /* Bring in data from switches */
62      3               BLOCK0=NOT INPUT(234);
                    /* Lockout temperature storage areas for update */
63      3               LOCKOUT=RQWAIT (.TEMP$LOCKOUT$EXCH,0);
                    /* Get raw data from analog converter */
64      3               Call RQSEND (.RQAIEX,.CONVERT);
65      3               MSG$PTR=RQWAIT(.A$D$EXCH,0);
                    /* Temperature calibrate procedure */
66      3               MSG$PTR=RQACPT(.TEMP$CALIBRATE);
67      3               If MSG$PTR <> 0
                        then do;
69      4                   k=0;
70      4                   Do while (TABLE(k)<>CALTEMP.CAL AND
                                k<255);
71      5                       k=k+1;
72      5                   end;
73      4                   Do n=0 to 3;
74      5                       OFFSET(n)=(TEMP(n)/16)-k;
75      5                   end;
76      4               end;
                    /* Convert data into engineering units */
77      3               Do n=0 to 3;
78      4                   If ((TEMP(n)/16)-OFFSET(n))>255
                            then TEMP(n)=0;
80      4                   else TEMP(n)=TABLE((TEMP(n)/16)-OFFSET(n));
81      4               end;
                    /* Release lockout of temperatures */
82      3               Call RQSEND (.TEMP$LOCKOUT$EXCH,LOCKOUT);
                    /* Compute average temperature */
```

```
83   3              Do n=0 to 3;
84   4                  T$AVERAGE(n)=(T$LAST(n)+TEMP(n))/2;
              /* Project temperatures into the future */
85   4                  If T$AVERAGE(n)>=T$LAST$AVERAGE(n)
                        then do;
87   5                     T$t5(n)=((T$AVERAGE(n)-T$LAST$AVERAGE(n))*5)
                                   +T$LAST$AVERAGE(n);
88   5                     T$t10(n)=((T$AVERAGE(n)-T$LAST$AVERAGE(n))*10)
                                    +T$LAST$AVERAGE(n);
89   5                  end;
90   4                  else do;
91   5                     T$t5(n)=T$LAST$AVERAGE(n)-((T$LAST$AVERAGE(n)
                                   -T$AVERAGE(n))*5);
92   5                     T$t10(n)=T$LAST$AVERAGE(n)-((T$LAST$AVERAGE(n)
                                    -T$AVERAGE(n))*10);
93   5                  end;
              /* Update stored data */
94   4                  T$LAST$AVERAGE(n)=T$AVERAGE(n);
95   4                  T$LAST(n)=TEMP(n);
              /* Test for active oven */
96   4                  MSG$PTR=RQWAIT (.CONSTANT$LOCKOUT$EXCH,0);
97   4                  If (((BLOCK0 AND OVEN$ON$MASK(n))<>0)
                        AND (TEMP(n)<>0))
                        then do;
99   5                     STATUS(n)=7;
100  5                     BLOCK2=BLOCK2 OR OVEN$RUN(n);
              /* Test for an intolerance condition */
101  5                     If SETPOINT(n)-TOLERANCE(n) < TEMP(n) AND
                              SETPOINT(n)+TOLERANCE(n) > TEMP(n)
                           then do;
103  6                        STATUS(n)=7;
104  6                        BLOCK1=BLOCK1 OR OVEN$IN$TOL(n);
105  6                     end;
106  5                     else BLOCK1=BLOCK1 AND NOT OVEN$IN$TOL(n);
              /* Test for a caution condition */
107  5                     If SETPOINT(n)-TOLERANCE(n) > T$t5(n) OR
                              SETPOINT(n)+TOLERANCE(n) < T$t5(n)
                           then do;
109  6                        STATUS(n)=14;
110  6                        BLOCK1=BLOCK1 OR OVEN$CAUTION(n);
111  6                     end;
112  5                     else BLOCK1=BLOCK1 AND NOT OVEN$CAUTION(n);
              /* Test for a danger condition */
113  5                     If SETPOINT(n)-TOLERANCE(n) > TEMP(n) OR
                              SETPOINT(n)+TOLERANCE(n) < TEMP(n)
                           then do;
115  6                        STATUS(n)=21;
116  6                        BLOCK2=BLOCK2 OR OVEN$DANGER(n);
117  6                     end;
118  5                     else BLOCK2=BLOCK2 AND NOT OVEN$DANGER(n);
              /* Handle control of heater elements */
119  5                     If SETPOINT(n) > T$t10(n)
                           then BLOCK3=BLOCK3 OR OVEN$HEATER(n);
121  5                     else BLOCK3=BLOCK3 AND NOT OVEN$HEATER(n);
122  5                  end;
123  4                  else do;
              /* Turn everything off when operator shuts off oven */
124  5                     BLOCK1=BLOCK1 AND NOT OVEN$IN$TOL(n);
125  5                     BLOCK1=BLOCK1 AND NOT OVEN$CAUTION(n);
126  5                     BLOCK3=BLOCK3 AND NOT OVEN$HEATER(n);
```

```
127   5              BLOCK2=BLOCK2 AND NOT OVEN$DANGER(n);
128   5              BLOCK2=BLOCK2 AND NOT OVEN$RUN(n);
129   5                STATUS(n)=0;
130   5            end;
131   4            Call RQSEND(.CONSTANT$LOCKOUT$EXCH,MSG$PTR);
132   4          end;

                 /* Output data to real world */
133   3            OUTPUT(232)=BLOCK1;
134   3            OUTPUT(233)=BLOCK2;
135   3            OUTPUT(234)=BLOCK2;
136   3          end;
137   2      end CONTROL$TASK;
138   1    end CONTROL$TASK$MODULE;

          MODULE INFORMATION:

                CODE AREA SIZE      = 0946H    2374D
                VARIABLE AREA SIZE  = 0054H      84D
                MAXIMUM STACK SIZE  = 0006H       6D
                235 LINES READ
                0 PROGRAM ERROR(S)


          END OF PL/M-80 COMPILATION



          $TITLE('CRT PARAMETER TASK')
          /**********************************************
          * This task is used to examine and update the *
          * temperature setpoints and tolerances for    *
          * each of the four ovens.                      *
          **********************************************/
1         UPDATE$TASK:
          Do;

          $Include (:F0:COMMON.ELT)
2    1  = DECLARE TRUE LITERALLY '0FFH';
3    1  = DECLARE FALSE LITERALLY '00H';
4    1  = DECLARE BOOLEAN LITERALLY 'BYTE';
5    1  = DECLARE FOREVER LITERALLY 'WHILE 1';
          $Include (:F0:MSGTYP.ELT)
6    1  = DECLARE DATA$TYPE LITERALLY '0',
        =         INT$TYPE LITERALLY '1',
        =         MISSED$INT$TYPE LITERALLY '2',
        =         TIME$OUT$TYPE LITERALLY '3',
        =         FS$REQ$TYPE LITERALLY '4',
        =         UC$REQ$TYPE LITERALLY '5',
        =         FS$NAK$TYPE LITERALLY '6',
        =         CNTRL$C$TYPE LITERALLY '7',
        =         READ$TYPE LITERALLY '8',
        =         CLR$RD$TYPE LITERALLY '9',
        =         LAST$RD$TYPE LITERALLY '10',
        =         ALARM$TYPE LITERALLY '11',
        =         WRITE$TYPE LITERALLY '12';
          $Include (:F0:MSG.ELT)
```

```
 7   1   =   DECLARE MSG$HDR LITERALLY '
         =     LINK ADDRESS,
         =     LENGTH ADDRESS,
         =     TYPE BYTE,
         =     HOME$EX ADDRESS,
         =     RESP$EX ADDRESS';
         =
 8   1   =   DECLARE MSG$DESCRIPTOR LITERALLY 'STRUCTURE(
         =     MSG$HDR,
         =     REMAINDER(1) BYTE)';
             $Include (:F0:THMSG.ELT)
 9   1   =   DECLARE TH$MSG LITERALLY 'STRUCTURE (
         =     MSGHDR,
         =     STATUS ADDRESS,
         =     BUFFER$ADR ADDRESS,
         =     COUNT ADDRESS,
         =     ACTUAL ADDRESS,
         =     REMAINDER(128) BYTE)';
10   1   =   DECLARE MIN$TH$MSG$LENGTH LITERALLY '17';
             $Include (:F0:CHAR.ELT)
         =
         =   /* SPECIAL ASCII CHARACTERS */
         =
11   1   =   DECLARE
         =     NULL          LITERALLY '00H',
         =     CONTROL$C     LITERALLY '03H',
         =     CONTROL$E     LITERALLY '05H',
         =     BELL          LITERALLY '07H',
         =     TAB           LITERALLY '09H',
         =     LF            LITERALLY '0AH',
         =     VT            LITERALLY '0BH',
         =     FF            LITERALLY '0CH',
         =     CR            LITERALLY '0DH',
         =     CONTROL$P     LITERALLY '10H',
         =     CONTROL$Q     LITERALLY '11H',
         =     CONTROL$R     LITERALLY '12H',
         =     CONTROL$S     LITERALLY '13H',
         =     CONTROL$X     LITERALLY '18H',
         =     CONTROL$Z     LITERALLY '1AH',
         =     ESC           LITERALLY '1BH',
         =     QUOTE         LITERALLY '22H',
         =     LCA           LITERALLY '61H',
         =     LCZ           LITERALLY '7AH',
         =     RUBOUT        LITERALLY '7FH';
         =
             $Include (:F0:SYNCH.EXT)
12   1   =   RQSEND:
         =     PROCEDURE (EXCHANGE$POINTER,MESSAGE$POINTER) EXTERNAL;
13   2   =       DECLARE (EXCHANGE$POINTER,MESSAGE$POINTER) ADDRESS;
         =
14   2   =     END RQSEND;
         =
15   1   =   RQWAIT:
         =     PROCEDURE (EXCHANGE$POINTER,DELAY) ADDRESS EXTERNAL;
16   2   =       DECLARE (EXCHANGE$POINTER,DELAY) ADDRESS;
         =
```

```
17   2   =      END RQWAIT;
         =
18   1   =   RQACPT:
         =      PROCEDURE (EXCHANGE$POINTER) ADDRESS EXTERNAL;
19   2   =      DECLARE EXCHANGE$POINTER ADDRESS;
         =
20   2   =      END RQACPT;
         =
21   1   =   RQISND:
         =      PROCEDURE (IED$PTR) EXTERNAL;
22   2   =      DECLARE IED$PTR ADDRESS;
         =
23   2   =      END RQISND;
24   1       Declare TEMP$CALIBRATE(5) address external;
25   1       Declare UPDATE$EXCH(5) address external;
26   1       Declare CRT$STATUS$EXCH(5) address external;
27   1       Declare COMP$EXCH(5) address external;
28   1       Declare CONSTANT$LOCKOUT$EXCH(5) address external;
29   1       Declare RQOUTX(5) address external;
30   1       Declare RQINPX(5) address external;
31   1       Declare WORDS$EXCH(5) address external;
32   1       Declare SETPOINT(4) address external;
33   1       Declare TOLERANCE(4) address external;
34   1       Declare BUFFER2 address;
35   1       Declare MSG$PTR address;
36   1       Declare MSG structure (
                  MSG$HDR,
                  STATUS address,
                  BUFFER$PTR address,
                  COUNT address,
                  ACTUAL address );
37   1       Declare CAL$TEMP structure (
                  MSG$HDR,
                  CAL address );
38   1       Declare UPD$MSG address;
39   1       Declare ENERGIZE based UPD$MSG structure (
                  MSG$HDR,
                  STATUS address,
                  BUFFER$PTR address,
                  COUNT address,
                  ACTUAL address );
40   1       Declare ENABLE$MSG structure (
                  MSG$HDR );
41   1       Declare BUFFER(80) byte;
42   1       Declare OVEN byte;
43   1         DEC$REP:
                  Procedure (SOURCE,TARGET) external;
44   2            Declare (SOURCE,TARGET) address;
45   2       end DEC$REP;
46   1         ASC$2$BINARY:
                     Procedure (SOURCE,TARGET,SIZE) byte external;
47   2            Declare (SOURCE,TARGET) address;
48   2            Declare SIZE byte;
49   2       end ASC$2$BINARY;
50   1         Declare MSG$1(20) byte data (
                  ESC,'E','ENTER OVEN NUMBER-');
```

```
51   1             Declare MSG$2(28) byte data (
                       CR,LF,
                       'ENTER NEW SETPOINT-',
                       'XXXX.X-' );
52   1             Declare MSG$3(29) byte data (
                       CR,LF,
                       'ENTER NEW TOLERANCE-',
                       'XXXX.X-' );
53   1             Declare CALMSG(12) byte data (
                       'TEMPERATURE-' );
54   1             Declare MSG$4(62) byte data (
                       CR,LF,
                       '(STATUS-(S), PARAMETERS-(P), CALIBRATE-(C))',
                       CR,LF,
                       'ENTER REQUEST-' );
55   1             Declare WAIT literally 'MSG$PTR=';
56   1             Declare FOR literally 'RQWAIT';
57   1             Declare START literally 'CALL';
58   1             Declare TASK literally 'RQSEND';
59   1          UPDATE:
             Procedure public;
                 /* Initialize task at start-up time */
60   2             Do forever;
61   3             MSG.RESP$EX=.COMP$EXCH;
                 /* Wait for request to enter task */
62   3                 UPD$MSG=RQWAIT (.UPDATE$EXCH,0);
                 /* Get desired oven number from operator */
63   3                 RQST$OVEN:
                       MSG.BUFFER$PTR=.MSG$1;
64   3                 MSG.TYPE=WRITE$TYPE;
65   3                 MSG.COUNT=20;
66   3                 Start task (.RQOUTX,.MSG);
67   3                 Wait for (.COMP$EXCH,0);
                 /* ...Input new number */
68   3                 MSG.BUFFER$PTR=.BUFFER;
69   3                 MSG.COUNT=255;
70   3                 MSG.TYPE=CLR$RD$TYPE;
71   3                 Start task (.RQINPX,.MSG);
72   3                 Wait for (.COMP$EXCH,0);
73   3                 OVEN=(BUFFER(0) AND 07H)-1;
74   3                 If OVEN >3 then go$to RQST$OVEN;
                 /* Display request and current setpoint */
76   3                 GET$TEMP:
                       Call move (28,.MSG$2,.BUFFER);
77   3                 Call DEC$REP (.SETPOINT(oven),.BUFFER+21);
78   3                 MSG.TYPE=WRITE$TYPE;
79   3                 MSG.COUNT=28;
80   3                 Start task (.RQOUTX,.MSG);
81   3                 Wait for (.COMP$EXCH,0);
                 /* ... Input new setpoint */
82   3                 MSG.TYPE=CLR$RD$TYPE;
83   3                 Start task (.RQINPX,.MSG);
84   3                 Wait for (.COMP$EXCH,0);
85   3                 If ASC$2$BINARY(.BUFFER,.BUFFER2,1)=0 OR BUFFER2 > 700
                       then go$to GET$TEMP;
87   3                 If BUFFER2 <> 0
                       then do;
89   4                     Wait for (.CONSTANT$LOCKOUT$EXCH,0);
90   4                     SETPOINT(oven)=BUFFER2;
91   4                     Start task (.CONSTANT$LOCKOUT$EXCH,MSG$PTR);
```

```
92   4           end;
              /* Display request and current tolerance */
93   3           GET$TOL:
                 Call move (29,.MSG$3,.BUFFER);
94   3           Call DEC$REP (.TOLERANCE(oven),.BUFFER+22);
95   3           MSG.TYPE=WRITE$TYPE;
96   3           MSG.COUNT=29;
97   3           Start task (.RQOUTX,.MSG);
98   3           Wait for (.COMP$EXCH,0);
              /* ...Input new tolerance */
99   3           MSG.TYPE=CLR$RD$TYPE;
100  3           Start task (.RQINPX,.MSG);
101  3           Wait for (.COMP$EXCH,0);
102  3           If ASC$2$BINARY(.BUFFER,.BUFFER2,1)=0 OR BUFFER2 > 700
                 then go$to GET$TOL;
104  3           If BUFFER2 <> 0
                 then do;
105  4               Wait for (.CONSTANT$LOCKOUT$EXCH,0);
107  4               TOLERANCE(oven)=BUFFER2;
108  4               Start task (.CONSTANT$LOCKOUT$EXCH,MSG$PTR);
109  4           end;
              /* Ask operator if he is finished */
110  3           REQ$NEXT:
                 MSG.TYPE=WRITE$TYPE;
111  3           MSG.COUNT=62;
112  3           MSG.BUFFER$PTR=.MSG$4;
113  3           Start task (.RQOUTX,.MSG);
114  3           Wait for (.COMP$EXCH,0);
              /* ...Get his response */
115  3           MSG.TYPE=CLR$RD$TYPE;
116  3           MSG.BUFFER$PTR=.BUFFER;
117  3           Start task (.RQINPX,.MSG);
118  3           Wait for (.COMP$EXCH,0);
119  3           If (BUFFER(0) <>'S' AND BUFFER(0) <>'P'
                  AND BUFFER(0) <> 'C')
                 then go$to REQ$NEXT;
121  3           If BUFFER(0)='P'
                    then go$to RQST$OVEN;
123  3           If BUFFER(0)='C'
                 then do;
125  4               GET$CAL:
                     MSG.TYPE=WRITE$TYPE;
126  4               MSG.COUNT=12;
127  4               MSG.BUFFER$PTR=.CALMSG;
128  4               Start task (.RQOUTX,.MSG);
129  4               Wait for (.COMP$EXCH,0);
130  4               MSG.TYPE=CLR$RD$TYPE;
131  4               MSG.BUFFER$PTR=.BUFFER;
132  4               Start task (.RQINPX,.MSG);
133  4               Wait for (.COMP$EXCH,0);
134  4               If ASC$2$BINARY(.BUFFER,.BUFFER2,1) =0
                        OR BUFFER2>350 OR BUFFER2<200
                     then go$to GET$CAL;
136  4               CAL$TEMP.CAL=BUFFER2;
137  4               Call RQSEND (.TEMP$CALIBRATE,.CAL$TEMP);
138  4           end;
```

```
                    MODULE INFORMATION:
                          CODE AREA SIZE    = 03C3H    963D
                          VARIABLE AREA SIZE = 007CH    124D
                          MAXIMUM STACK SIZE = 0004H      4D
                          264 LINES READ
                          0 PROGRAM ERROR(S)
                    END OF PL/M-80 COMPILATION


139    3              ENERGIZE.TYPE=100;
140    3              Start task (.CRT$STATUS$EXCH,UPD$MSG);

141    3          end;
142    2        end UPDATE;
143    1    end UPDATE$TASK;




                    $TITLE('CRT UPDATE TASK')
                    /********************************************
                    * This task is utilized to update the CRT ter- *
                    * minal display with the current operating par- *
                    * ameters. It will be entered upon sytem start- *
                    * up, upon operator request, or when a problem  *
                    * exists with any of the activated ovens.       *
                    ********************************************/
1                   CRT$DATA$MODULE:
                    Do;
                    $INCLUDE(:F0:SYNCH.EXT)
2     1    =        RQSEND:
      =              PROCEDURE (EXCHANGE$POINTER,MESSAGE$POINTER) EXTERNAL;
3     2    =           DECLARE (EXCHANGE$POINTER,MESSAGE$POINTER) ADDRESS;
      =
4     2    =        END RQSEND;
      =
5     1    =        RQWAIT:
      =              PROCEDURE (EXCHANGE$POINTER,DELAY) ADDRESS EXTERNAL;
6     2    =           DECLARE (EXCHANGE$POINTER,DELAY) ADDRESS;
      =
7     2    =        END RQWAIT;
      =
8     1    =        RQACPT:
      =              PROCEDURE (EXCHANGE$POINTER) ADDRESS EXTERNAL;
9     2    =           DECLARE EXCHANGE$POINTER ADDRESS;
      =
10    2    =        END RQACPT;
      =
11    1    =        RQISND:
      =              PROCEDURE (IED$PTR) EXTERNAL;
12    2    =           DECLARE IED$PTR ADDRESS;
      =
13    2    =        END RQISND;
                    $INCLUDE (:F0:MSGTYP.ELT)
14    1    =        DECLARE DATA$TYPE LITERALLY '0',
```

```
                 =            INT$TYPE LITERALLY '1',
                 =            MISSED$INT$TYPE LITERALLY '2',
                 =            TIME$OUT$TYPE LITERALLY '3',
                 =            FS$REQ$TYPE LITERALLY '4',
                 =            UC$REQ$TYPE LITERALLY '5',
                 =            FS$NAK$TYPE LITERALLY '6',
                 =            CNTRL$C$TYPE LITERALLY '7',
                 =            READ$TYPE LITERALLY '8',
                 =            CLR$RD$TYPE LITERALLY '9',
                 =            LAST$RD$TYPE LITERALLY '10',
                 =            ALARM$TYPE LITERALLY '11',
                 =            WRITE$TYPE LITERALLY '12';
                        $INCLUDE (:F0:EXCH.ELT)
15      1        =       DECLARE EXCHANGE$DESCRIPTOR LITERALLY 'STRUCTURE (
                 =         MESSAGE$HEAD ADDRESS,
                 =         MESSAGE$TAIL ADDRESS,
                 =         TASK$HEAD ADDRESS,
                 =         TASK$TAIL ADDRESS,
                 =         EXCHANGE$LINK ADDRESS)';
                        $INCLUDE (:F0:COMMON.ELT)
16      1        =       DECLARE TRUE LITERALLY '0FFH';
17      1        =       DECLARE FALSE LITERALLY '00H';
18      1        =       DECLARE BOOLEAN LITERALLY 'BYTE';
19      1        =       DECLARE FOREVER LITERALLY 'WHILE 1';
                        $INCLUDE (:F0:MSG.ELT)
20      1        =       DECLARE MSG$HDR LITERALLY '
                 =         LINK ADDRESS,
                 =         LENGTH ADDRESS,
                 =         TYPE BYTE,
                 =         HOME$EX ADDRESS,
                 =         RESP$EX ADDRESS';
                 =
21      1        =       DECLARE MSG$DESCRIPTOR LITERALLY 'STRUCTURE(
                 =         MSG$HDR,
                 =         REMAINDER(1) BYTE)';
                        $INCLUDE (:F0:CHAR.ELT)
                 =
                 =       /* SPECIAL ASCII CHARACTERS */
                 =
22      1        =       DECLARE
                 =          NULL           LITERALLY '00H',
                 =          CONTROL$C      LITERALLY '03H',
                 =          CONTROL$E      LITERALLY '05H',
                 =          BELL           LITERALLY '07H',
                 =          TAB            LITERALLY '09H',
                 =          LF             LITERALLY '0AH',
                 =          VT             LITERALLY '0BH',
                 =          FF             LITERALLY '0CH',
                 =          CR             LITERALLY '0DH',
                 =          CONTROL$P      LITERALLY '10H',
                 =          CONTROL$Q      LITERALLY '11H',
                 =          CONTROL$R      LITERALLY '12H',
                 =          CONTROL$S      LITERALLY '13H',
                 =          CONTROL$X      LITERALLY '18H',
                 =          CONTROL$Z      LITERALLY '1AH',
                 =          ESC            LITERALLY '1BH',
                 =          QUOTE          LITERALLY '22H',
```

```
           =      LCA              LITERALLY '61H',
           =      LCZ              LITERALLY '7AH',
           =      RUBOUT           LITERALLY '7FH';
           =
                  $INCLUDE (:F0:THMSG.ELT)
  23    1  =      DECLARE TH$MSG LITERALLY 'STRUCTURE (
           =      MSGHDR,
           =      STATUS ADDRESS,
           =      BUFFER$ADR ADDRESS,
           =      COUNT ADDRESS,
           =      ACTUAL ADDRESS,
           =      REMAINDER(128) BYTE)';
  24    1  =      DECLARE MIN$TH$MSG$LENGTH LITERALLY '17';
  25    1         Declare HOME literally '1BH,48H';
  26    1         Declare L1$IMAGE(90) byte data (
                  Home,Lf,Lf,Lf,Lf,Lf,
                  'TEMPERATURE          ',
                  '                     ',
                  '                     ',
                  '                     ',
                  '                    ',
                  'DEGREES C.' );
  27    1         Declare L2$IMAGE(92) byte data (
                      Home,Lf,Lf,Lf,Lf,Lf,Lf,Lf,
                      'SETPOINT             ',
                      '                     ',
                      '                     ',
                      '                     ',
                      '                    ',
                      'DEGREES C.' );
  28    1         Declare L3$IMAGE(94) byte data (
                      Home,Lf,Lf,Lf,Lf,Lf,Lf,Lf,Lf,Lf,
                      'TOLERANCE            ',
                      '                     ',
                      '                     ',
                      '                     ',
                      '                    ',
                      'DEGREES C.' );
  29    1         Declare L4$IMAGE(75) byte data (
                      Home,Lf,Lf,Lf,Lf,Lf,Lf,Lf,Lf,Lf,Lf,Lf,
                      'STATUS               ',
                      '   OFF               ',
                      '   OFF               ',
                      '   OFF               ',
                      '   OFF    ' );
  30    1         Declare CRT$HDR(168) byte data (
                      1BH,45H,'                                      ',
                      'OVEN STATUS DISPLAY',
                      Cr,Lf,Lf,'                        ',
                        'OVEN-1           ',
                      'OVEN-2           ',
                      'OVEN-3           ',
                      'OVEN-4',
```

```
                     Cr,Lf,Lf,Lf,Lf,Lf,Lf,Lf,Lf,Lf,Lf,Lf,Lf,Lf,Lf,Lf,Lf,Lf,Lf,
               Lf,
                     'TYPE ESCAPE TO ADJUST SETPOINTS' );
31    1        Declare BELLS(4) byte data (
                   Bell,Bell,Bell,Bell );
32    1        Declare MESSAGES(35) byte data (
                   '  OFF  ',
                   '  OK   ',
                   'CAUTION',
                   ' ALARM ',
                   '       ' );
33    1          Declare DISPLAY$PTR1(4) address data (
                   .WORK$BUFF+23,
                   .WORK$BUFF+36,
                   .WORK$BUFF+49,
                   .WORK$BUFF+62 );
34    1        Declare DISPLAY$PTR2(4) address data (
                   .WORK$BUFF+25,
                   .WORK$BUFF+38,
                   .WORK$BUFF+51,
                   .WORK$BUFF+64 );
35    1        Declare DISPLAY$PTR3(4) address data (
                   .WORK$BUFF+27,
                   .WORK$BUFF+40,
                   .WORK$BUFF+53,
                   .WORK$BUFF+66 );
36    1        Declare DISPLAY$PTR4(4) address data (
                   .WORKBUFF+30,
                   .WORKBUFF+43,
                   .WORKBUFF+56,
                   .WORKBUFF+69 );
37    1        Declare MSG$PTR address;
38    1        Declare MSG based MSG$PTR structure (
                   MSG$HDR,
                   COUNT address );
39    1        Declare STARTER(3) structure (
                   MSG$HDR );
40    1        Declare READ structure (
                   MSG$HDR,
                   STATUS address,
                   BUFFER$PTR address,
                   COUNT address,
                   ACTUAL address );
41    1        Declare DISPLAY$TEMP(4) structure (
                   UPPER address,
                   LOWER address );
42    1        Declare DISPLAY$SET(4) structure (
                   LOWER address,
                   UPPER address );
43    1        Declare DISPLAY$TOL(4) structure (
                   LOWER address,
                   UPPER address );
```

```
44    1          Declare OVEN$ON(4) byte data (
                 01H,02H,04H,08H );
45    1        Declare OVEN$CAUTION(4) byte data (
                 10H,20H,40H,80H );
46    1          Declare CRT structure (
                 MSG$HDR,
                 STATUS address,
                 BUFFER$PTR address,
                 COUNT address,
                 ACTUAL address );
47    1          Declare CRTLOCK structure (MSG$HDR);
48    1          Declare CRT$DISPLAY$LOCK(5) address external;
49    1        Declare TEMP$LOCKOUT$EXCH(5) address external;
50    1        Declare CONSTANT$LOCKOUT$EXCH(5) address external;
51    1        Declare CRT$EXCH(5) address external;
52    1        Declare CRT$STATUS$EXCH(5) address external;
53    1        Declare DUMMY$EXCH(5) address external;
54    1        Declare READ$BUFFER$EXCH(5) address external;
55    1        Declare UPDATE$EXCH(5) address external;
56    1        Declare RQINPX(5) address external;
57    1        Declare RQOUTX(5) address external;
58    1        Declare RQWAKE(5) address external;
59    1        Declare RQL7EX(5) address external;
60    1        Declare RQL6EX(5) address external;
61    1        Declare RQDBUG(5) address external;
62    1        Declare RQALRM(5) address external;
63    1          Declare TEMP(4) address external;
64    1        Declare DISP$TEMP(4) address;
65    1        Declare SETPOINT(4) address external;
66    1        Declare DISP$SETPNT(4) address;
67    1        Declare TOLERANCE(4) address external;
68    1        Declare DISP$TOL(4) address;
69    1        Declare STATUS(4) byte external;
70    1        Declare DISP$STAT(4) byte;
71    1        Declare (BLOCK1,BLOCK2) byte external;
72    1        Declare WORK$BUFF(170) byte;
73    1        Declare BUFFER$A(70) byte;
74    1        Declare (CHANGE,n,ALARM,NEW,BLANKER) byte;
75    1          Declare START literally 'call';
76    1        Declare TASK literally 'rqsend';
77    1        Declare WAIT literally 'msg$ptr=';
78    1        Declare For literally 'rqwait';
79    1          DEC$REP:
                 Procedure(SOURCE,TARGET) external;
80    2          Declare (SOURCE,TARGET) address;
81    2        end DEC$REP;
```

```
82    1         CRT$DATA$TASK:
                Procedure public;
                /* Initialize system at start-up time */
83    2           Start task (.TEMP$LOCKOUT$EXCH,.STARTER(0));
84    2           Start task (.CONSTANT$LOCKOUT$EXCH,.STARTER(1));
85    2           STARTER(2).TYPE=100;
86    2           Start task (.CRT$STATUS$EXCH,.STARTER(2));
87    2           CRT.RESP$EX=.CRT$EXCH;
                /* Perform main CRT wait */
88    2           Do forever;
89    3               Wait for (.DUMMY$EXCH,10);
90    3               Wait for (.CRT$STATUS$EXCH,0);
91    3               If MSG.TYPE=255
                        then ALARM=1;
93    3               else ALARM=0;
                /* Output heading */
94    3               If (MSG.TYPE=100 OR MSG.TYPE=255)
                        then do;
96    4                   If ALARM=0
                            then call RQSEND(.CRT$DISPLAY$LOCK,.CRTLOCK);
98    4                   CRT.TYPE=WRITE$TYPE;
99    4                   CRT.COUNT=167;
100   4                   CRT.BUFFER$PTR=.WORK$BUFF;
101   4                   READ.TYPE=CLR$RD$TYPE;
102   4                   READ.COUNT=255;
103   4                   READ.RESP$EX=.READ$BUFFER$EXCH;
104   4                   READ.BUFFER$PTR=.BUFFERA;
105   4                   If ALARM=0
                            then start task (.RQINPX,.READ);
107   4                   Call move (82,.CRT$HDR,.WORK$BUFF);
108   4                   Call move (86,.CRT$HDR+82,.WORK$BUFF+82);
109   4                   Start task (.RQOUTX,.CRT);
110   4                   Wait for (.CRT$EXCH,0);
111   4                   NEW=1;
112   4               end;
                /* Test for change in temperature of any oven */
113   3               CHANGE=0;
114   3               Wait for (.TEMP$LOCKOUT$EXCH,0);
115   3               Do n=0 to 3;
116   4                   If TEMP(n)<>DISP$TEMP(n)
                            then CHANGE=1;
118   4               end;
119   3               Call move (8,.TEMP,.DISP$TEMP);
120   3               Start task (.TEMP$LOCKOUT$EXCH,MSG$PTR);
                /* When a change exists build new line */
121   3               If CHANGE OR NEW
                        then do;
123   4                   Call move (90,.L1$IMAGE,.WORK$BUFF);
124   4                   Do n=0 to 3;
125   5                       Call DEC$REP(.DISP$TEMP(n),DISPLAY$PTR1(n));
126   5                   end;
                /* Output new temperature line to CRT */
127   4                   CRT.TYPE=WRITE$TYPE;
128   4                   CRT.COUNT=87;
```

```
129  4              Start task (.RQOUTX,.CRT);
130  4              Wait for (.CRT$EXCH,0);
131  4          end;
         /* Test for change in oven setpoints */
132  3          CHANGE=0;
133  3          Wait for (.CONSTANT$LOCKOUT$EXCH,0);
134  3          Do n=0 to 3;
135  4              If SETPOINT(n)<>DISP$SETPNT(n)
                    then CHANGE=1;
137  4          end;
138  3          Call move (8,.SETPOINT,.DISP$SETPNT);
139  3          Start task (.CONSTANT$LOCKOUT$EXCH,MSG$PTR);
         /* Build new line when a change was detected */
140  3          If CHANGE OR NEW
                then do;
142  4              Call move (92,.L2$IMAGE,.WORKBUFF);
143  4              Do n=0 to 3;
144  5                  Call DEC$REP(.DISP$SETPNT(n),DISPLAY$PTR2(n));
145  5              end;
         /* Output setpoint line */
146  4              CRT.TYPE=WRITE$TYPE;
147  4              CRT.COUNT=89;
148  4              CRT.BUFFER$PTR=.WORKBUFF;
149  4              Start task (.RQOUTX,.CRT);
150  4              Wait for (.CRT$EXCH,0);
151  4          end;
         /* Test for change in tolerance line */
152  3          CHANGE=0;
153  3          Wait for (.CONSTANT$LOCKOUT$EXCH,0);
154  3          Do n=0 to 3;
155  4              If TOLERANCE(n)<>DISP$TOL(n)
                    then CHANGE=1;
157  4          end;
158  3          Call move (8,.TOLERANCE,.DISP$TOL);
159  3          Start task (.CONSTANT$LOCKOUT$EXCH,MSG$PTR);
         /* When change is found, build new line */
160  3          If CHANGE OR NEW
                then do;
162  4              Call move (94,.L3$IMAGE,.WORK$BUFF);
163  4              Do n=0 to 3;
164  5                  Call DEC$REP(.DISP$TOL(n),DISPLAY$PTR3(n));
165  5              end;
         /* Output tolerance line */
166  4              CRT.TYPE=WRITE$TYPE;
167  4              CRT.COUNT=91;
168  4              CRT.BUFFER$PTR=.WORKBUFF;
169  4              Start task (.RQOUTX,.CRT);
170  4              Wait for (.CRT$EXCH,0);
171  4          end;
         /* Build status message */
172  3          CHANGE=0;
173  3          Wait for (.CONSTANT$LOCKOUT$EXCH,0);
174  3          Do n=0 to 3;
175  4              If STATUS(n)<>DISP$STAT(n)
                    then CHANGE=1;
```

```
177   4             end;
178   3             Call move (4,.STATUS,.DISP$STAT);
179   3             Start task (.CONSTANT$LOCKOUT$EXCH,MSG$PTR);
             /* Output to display */
180   3             If CHANGE OR NEW
                    then do;
182   4                 Call move (75,.L4IMAGE,.WORK$BUFF);
183   4                 Do n=0 to 2;
184   5                     Call move(7,.MESSAGES+DISP$STAT(n),DISPLAY$PTR4(
          n));
185   5                 end;
186   4                 CRT.COUNT=76;
187   4                 Start task (.RQOUTX,.CRT);
188   4                 Wait for (.CRT$EXCH,0);
189   4             end;
             /* test for request to exit this mode */
190   3             MSG$PTR=RQACPT (.READ$BUFFER$EXCH);
191   3             If ALARM=0
                    then do;
193   4                 If (MSG$PTR <> 0 and BUFFERA(0) = 1BH)
                        then do;
195   5                     MSG$PTR=RQWAIT(.CRT$DISPLAY$LOCK,0);
196   5                     start task (.UPDATE$EXCH,MSG$PTR);
197   5                 end;
198   4                 else do;
199   5                     If MSG$PTR=0
                            then STARTER(2).TYPE=200;
201   5                     else STARTER(2).TYPE=100;
202   5                     Start task (.CRT$STATUS$EXCH,.STARTER(2));
203   5                     NEW=0;
204   5                 end;
205   4             end;
206   3         end;
207   2     end CRT$DATA$TASK;
208   1 end CRT$DATA$MODULE;


MODULE INFORMATION:
    CODE AREA SIZE     = 0720H    1824D
    VARIABLE AREA SIZE = 0189H     393D
    MAXIMUM STACK SIZE = 0004H       4D
    388 LINES READ
    0 PROGRAM ERROR(S)

END OF PL/M-80 COMPILATION
```

```
                $TITLE('ASCII STRING TO FIXED BINARY')
                /***********************************************
                * This program converts an ASCII string into a fix- *
                * ed point binary number.  The fixed decimal point   *
                * is determined by the parameter passed in SIZE.     *
                ************************************************/
1               ASC$2$BINARY$MODULE:
                Do;
                /* SPECIAL ASCII CHARACTERS */
2        1      DECLARE
                   NULL           LITERALLY '00H',
                   CONTROL$C      LITERALLY '03H',
                   CONTROL$E      LITERALLY '05H',
                   BELL           LITERALLY '07H',
                   TAB            LITERALLY '09H',
                   LF             LITERALLY '0AH',
                   VT             LITERALLY '0BH',
                   FF             LITERALLY '0CH',
                   CR             LITERALLY '0DH',
                   CONTROL$P      LITERALLY '10H',
                   CONTROL$Q      LITERALLY '11H',
                   CONTROL$R      LITERALLY '12H',
                   CONTROL$S      LITERALLY '13H',
                   CONTROL$X      LITERALLY '18H',
                   CONTROL$Z      LITERALLY '1AH',
                   ESC            LITERALLY '1BH',
                   QUOTE          LITERALLY '22H',
                   LCA            LITERALLY '61H',
                   LCZ            LITERALLY '7AH',
                   RUBOUT         LITERALLY '7FH';

3        1      ASC$2$BINARY:
                Procedure (SRC$PTR,TRGT$PTR,SIZE) byte public;
4        2          Declare (SRC$PTR,TRGT$PTR) address;
5        2          Declare (SOURCE based SRC$PTR)(80) byte;
6        2          Declare RESULT based TRGT$PTR address;
7        2          Declare (N,SIZE,K,DP,DIGITS,VALID) byte;
8        2          Declare POWER(6) address data (
                        0,1,10,100,1000,10000 );
                /* Find location of decimal point */
9        2          n=0;
10       2          Do while SOURCE(n)<>'.' AND SOURCE(n)<>CR
                            AND SOURCE(n)<>LF;
11       3              n=n+1;
12       3            end;
13       2          DP=n;
                /* Provide correct number of digits to right of decimal */
14       2          Do n=0 to SIZE;
15       3              SOURCE(DP+n)=SOURCE(DP+n+1);
16       3              If SOURCE(DP+n)>39H OR SOURCE(DP+n)<30H
                        then do k=n to SIZE;
18       4                  SOURCE(DP+k)='0';
19       4              end;
```

```
20   3           end;
                 /* Mark end of string */
21   2             DIGITS=DP+SIZE;
                 /* Test for all valid characters */
22   2             VALID=1;
23   2             Do n=0 to DIGITS;
24   3                 If SOURCE(n)>39H OR SOURCE(n)<30H
                       then VALID=0;
26   3             end;
27   2             If DIGITS>5
                   then VALID=0;
                 /* Convert data to binary and store */
29   2             n=0;
30   2             If VALID=1
                   then do;
32   3                 RESULT=0;
33   3                 Do while DIGITS > 0;
34   4                     RESULT=RESULT+((
                                   SOURCE(n) AND 0FH) * POWER(DIGITS));
35   4                     n=n+1;
36   4                     DIGITS=DIGITS-1;
37   4                 end;
38   3             end;
                 /* Return to calling program */
39   2             Return VALID;
40   2           end ASC$2$BINARY;
41   1         end ASC$2$BINARY$MODULE;
```

MODULE INFORMATION:

```
    CODE AREA SIZE      = 0178H     376D
    VARIABLE AREA SIZE  = 000AH      10D
    MAXIMUM STACK SIZE  = 0004H       4D
    80 LINES READ
    0 PROGRAM ERROR(S)
```

END OF PL/M-80 COMPILATION

```
                STITLE('COMMON VARIABLE STORAGE')
                /***********************************************
                * This module contains those variables common to  *
                * multiple tasks in the oven control application. *
                ***********************************************/
    1           VARIABLE$STORAGE:
                Do;
    2     1     Declare SETPOINT(4) address public;
    3     1     Declare TOLERANCE(4) address public;
    4     1     Declare TEMP(4) address public;
    5     1     Declare STATUS(4) byte public;
    6     1     Declare BLOCK0 byte public;
    7     1     Declare BLOCK1 byte public;
    8     1     Declare BLOCK2 byte public;
    9     1     Declare BLOCK3 byte public;
   10     1     end VARIABLE$STORAGE;
```

MODULE INFORMATION:

```
    CODE AREA SIZE      = 0000H       0D
    VARIABLE AREA SIZE = 0020H       32D
    MAXIMUM STACK SIZE = 0000H       0D
    16 LINES READ
    0 PROGRAM ERROR(S)
```

END OF PL/M-80 COMPILATION

```
                    $TITLE('WORD TO ASCII CONVERSION')
                    /*****************************************************
                    * This routine converts a fixed point word in mem-  *
                    * ory into a 4 digit plus 1 decimal ASCII display-   *
                    * able number.  Zero blanking is included.           *
                    *****************************************************/
   1                DEC$REP$MODULE:
                    Do;

   2     1          DEC$REP:
                    Procedure (SOURCE,TARGET) public ;
   3     2              Declare (SOURCE,TARGET) address;
   4     2              Declare ANSWR(5) byte;
   5     2              Declare (DISPLAY based TARGET)(5) byte;
   6     2              Declare NUMBER based SOURCE structure (
                            ELEMENT address );
   7     2              Declare N byte;
   8     2              Declare CALC(5) address;
                       /* Initialize */
   9     2              Do n=0 to 4;
  10     3                  ANSWR(n)='0';
  11     3              end;
  12     2              CALC(0)=NUMBER.ELEMENT;
                       /* Convert to ASCII */
  13     2              Do n=1 to 5;
  14     3                  CALC(n)=CALC(n-1)/10;
  15     3                  ANSWR(5-n)=(CALC(n-1) mod 10) + 30H;
  16     3              end;
                       /* Perform zero blanking */
  17     2              Do n=0 to 3;
  18     3                  If ANSWR(n)<>'0'
                           then n=4;
  20     3                  else ANSWR(n)=' ';
  21     3              end;
                       /* Format with decimal point */
  22     2              Call move (4,.ANSWR,TARGET);
  23     2              DISPLAY(4)='.';
  24     2              DISPLAY(5)=ANSWR(4);
  25     2          end DEC$REP;
  26     1      end DEC$REP$MODULE;


MODULE INFORMATION:

    CODE AREA SIZE      = 00EEH      238D
    VARIABLE AREA SIZE  = 0014H       20D
    MAXIMUM STACK SIZE  = 0004H        4D
    40 LINES READ
    0 PROGRAM ERROR(S)
END OF PL/M-80 COMPILATION
```

November 1979

Closed Loop Control Using
The iSBC 569/941™
Intelligent Digital
Processors

Peter Andersen
OEM Microcomputer Systems Applications

9803104

Closed Loop Control Using the iSBC 569/941 Intelligent Digital Processors

Contents

I. INTRODUCTION

The utilization of computers to provide control or monitoring functions for industrial processes frequently results in complex computer systems. Distributing the control and processing intelligence throughout the control network reduces significantly the complexity of the system while increasing the reliability. The physical areas being controlled or monitored by each portion of the distributed system will generally consist of a relatively small number of I/O functions which are related by some control algorithm.

The Intel iSBC 569 Intelligent Digital Controller (IDC) and the iSBC 941 Industrial Digital Processor (IDP) are a part of the expanding line of Intel products which are oriented toward filling the requirements of these systems. This application note deals with the use of these devices to provide control of a closed loop system using a version of the PID control algorithm.

It is assumed that the reader is familiar with the basic concepts required to generate software and has had some experience with using a computer. This application note will then guide the reader through a typical application, explaining in detail the decisions which must be made in order to effectively utilize a microcomputer to provide a control solution.

The application which has been chosen is considered to be typical of the type which lends itself to control. The mechanical aspects of the application will be explained so that the user not familiar with the particular machinery will be able to understand the development. It will be seen that the techniques used will apply to any other specific application.

The emphasis of the note will be on the use and implementation of the hardware and software features of the digital processor and controller. The actual PID control algorithm will not be developed in this application note.

Reasons for Intelligent Boards

The advent of microcomputers and the resulting trend toward utilizing these devices to control processes has resulted in many cases where the overall system performance has deteriorated because of the demands placed on the processor.

In these applications, the computer has become overburdened with control algorithms, alarm detection, communications, and the many other tasks required of it. The processor can be interrupted by time dependent tasks to the point where other processing tasks can not be completed.

Presently, Intel provides two I/O expansion boards which are capable of handling portions of the processing load which formally required processor time. These two devices are the iSBC 544 Intelligent Communications Controller and the iSBC 569 Intelligent Digital Controller. Tasks which involve communications or parallel digital I/O can now be offloaded without requiring valuable processor time. These boards can issue interrupts to the master or host processor if interaction with other processes or devices is required. This technique greatly increases system throughput by offloading the other bus master processors and by minimizing traffic on the Multibus system bus.

In some cases, it will be found that the intelligent controller can function to control the process in a stand-alone environment, providing a more functional, low cost control system.

The concept of offloading the processor of its input/output tasks can be developed on the iSBC 569 controller through the use of slave processors which may be installed on the board to assist the host. The result is the ability to provide up to four processors on a single intelligent slave I/O board by using the concept of slave processors.

The On-Board Slave Concept

The utilization of the iSBC 569 controller is enhanced through the use of On Board Slave processors (OBS). These devices distribute the system intelligence and offload the processor on the intelligent controller. They can provide custom digital interfaces with the various devices which may be connected to the I/O ports of the controller. The OBS device allows a designer to fully specify his control/interface algorithm in the peripheral chip without relying on the master processor. Three types of OBS compatible devices are available from Intel. These are: 1) Industrial Processors, 2) Standard UPI devices, and 3) UPI 8741A for custom applications. By combining the

devices in various combinations, optimum solutions can be generated for different control applications.

Before proceeding, we should cover the general characteristics of the OBS devices available for use in conjunction with the iSBC 569 controller. It will be seen that careful selection of the proper I/O controller chip can reduce significantly the design effort required to provide a control solution.

II. BASIC UNIVERSAL PERIPHERAL INTERFACE DISCUSSION

With the introduction of the Universal Peripheral Interface, Intel has expanded the intelligent peripheral concept by providing an intelligent controller that is fully user programmable. The 8741A is a complete single-chip microcomputer which connects directly to a master processor data bus.

To fully understand the techniques used by the UPI 8741A devices, we must have a general knowledge of their characteristics. Only then will we feel comfortable in implementing a design which uses the components.

Hardware Features

Each Universal Peripheral Interface has 1K bytes of program storage plus 64 bytes of RAM memory for data storage. It has a powerful, 8-bit CPU with a 2.5 μsec cycle time and two interrupts. Over 90 instructions are provided in its instruction set. Most instructions are single byte and single cycle and none are more than two bytes long. These instructions are optimized for bit manipulation and I/O operations. Special instructions are included to allow binary or BCD arithmetic operations, table lookup routines, loop counters, and N-way branch routines.

The chip's 8-bit interval timer/event counter can be used to generate complex timing sequences for control applications or it can count external events such as switch closures and position encoder pulses. Software timing loops can be simplified or eliminated by the interval timer. If enabled, an interrupt to the CPU can occur when the timer overflows.

Two 8-bit bidirectional I/O ports are included which are TTL compatible. Each of the 16 port lines can individually function as either input or output under software control.

The UPI microcomputer is fully supported with development tools. The combination of device features and Intel development support make the 8741A an ideal component for low-speed peripheral control applications.

Software Interface

The OBS communicates with the processor on the host board by means of data transfers between its registers and the host board's data bus. A communication protocol has been defined which provides a set of rules by which the processors may interact with each other. Two types of software protocol are currently defined. These are the "simple" and the "extended" protocol. Before attempting to utilize the OBS devices in an application, the concepts used for the communications must be fully understood.

When used on one of Intel's single board computers, the communication path is by means of the I/O ports on the host board. This means that two port addresses, an odd and an even location, are assigned to each OBS device. The even numbered port is used to transfer "data" between the processors. The odd numbered port is used to write commands into the OBS and to read its status. Each transfer between the host and the slave device consists of the movement of eight bits of information.

Four of the eight bits available in the status message have been given predefined functions. The bit will be set (logical 1) when the corresponding condition exists within the OBS device and will be reset (logical 0) when the condition does not exist. The functions of the four bits are:

Bit-0. Output Buffer Full (OBF).
This bit indicates that the OBS has placed information into the transfer register and that the information is available to the host processor. It can be read by performing an input operation from the even numbered port assigned to the particular OBS. When the data has been read, the bit will automatically be reset to indicate that no data is available. As we will see, this is one of the key features enabling efficient utilization of the host/

slave relationships on single board computers.

Bit-1. Input Buffer Full (IBF).

This bit is used to indicate that data has been placed into the input transfer register by the host device and that it has not yet been read by the slave. Data is transferred into the input register by means of the host performing an output to the even numbered port of the OBS. The bit will be reset when the device reads the data from the input transfer register into its accumulator. Data should only be output to the OBS when the IBF bit is reset!

Bit-2. F0 Flag.

Unlike the IBF and the OBF bits which are controlled by hardware, the F0 bit is controlled by the device software. The normal function of the flag is to provide a lockout to prevent the host from sending more data until the previous data has been processed or the operation is complete.

Bit-3. F1 is the Command/Data Flag.

It is automatically set when the host sends either a command (odd numbered port) or data (even numbered port). A logical 1 indicates that a command has been sent and a logical 0 indicates that data has been sent. This bit may also be cleared or toggled by the UPI software.

These bits will provide normal communications between the master and slave processors.

Figure 1 shows the sequence of operations which can be used by the host processor to establish communications with an OBS using the simple protocol. In Figure 1a, we see that all operations are initiated by the host. It will first verify that the IBF flag indicates that the input register is empty and available for receiving a command. The command is then sent to the odd numbered port. This command will inform the OBS that is to perform some task. The task may involve a requirement for more information to be sent to the controller and it may involve the controller returning some data to the host. Figure 1b shows the operations required for receiving data from the OBS.
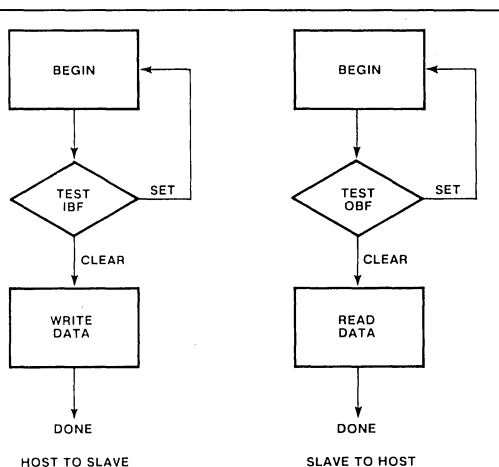
Figure 1. Simple Protocol

With these ideas in mind, we can move to a discussion of representative versions of the devices available for use on the IDC boards. We will then look at a typical application to see how they can actually be applied to solve a problem.

Standard Universal Peripheral Controllers

Intel presently manufactures three UPI controllers for non-industrial applications. These are:

1. 8278 Programmable Keyboard Interface
2. 8294 Data Encryption Unit
3. 8295 Dot Matrix Printer Controller

These devices offer an "off the shelf" solution to many applications which might be encountered.

The Intel 8278 is a general purpose programmable keyboard and display interface device. The keyboard portion can provide a scanned interface to 128-key contact or capacitive-coupled keyboards. The keys are fully debounced with N–key rollover and programmable error generation on multiple new key closures. Keyboard entries are stored in an 8-character FIF0 with overrun status indication when more than 8-characters have been entered. Key entries set an interrupt request output to the master CPU. The display portion of the 8278 provides a scanned display interface for LED, incandescent, and other popular display technologies. Both numeric displays and simple indicators may be used. The 8278 has a 16 x 4

display RAM which can be loaded or interrogated by the CPU. Both right entry calculator and left entry typewriter display formats are possible. Read and write of the display RAM can be done with auto-increment of the display RAM address.

The Intel 8294 Data Encryption Unit is designed to encode and decode 64-bit blocks of data using the algorithm specified in the Federal Information Processing Data Encryption Standard. The DEU operates on 64-bit test words using a 56-bit user specified key to produce 64-bit cipher words. The operation is reversible; if the cipher word is operated upon, the original test word is produced. Because the 8294 is compatible with the NBS encryption standard, it can be used in a variety of electronic funds transfer applications as well as other electronic banking and data handling applications where data must be encrypted.

Finally, the Intel 8295 Dot Matrix Printer Controller provides an interface to the LRC 7040 Series dot matrix impact printers. It may also be used as an interface to other similar printers. The chip may be used in a serial or parallel communication mode with the host processor. Furthermore, it provides internal buffering of up to 40 characters and contains a 7 x 7 matrix character generator which accommodates 64 ASCII characters.

Industrial Digital Processor

Intel produces the iSBC 941 Industrial Digital Processor (IDP) which is programmed to handle an assortment of typical industrial digital interfaces and transducers. The controller can function to provide any of the following:

1. Scan up to 16 inputs for a change of state.
2. Provide up to 8 gated one-shot outputs.
3. Provide eight gated outputs with programmable pulse widths and periods.
4. Provide monitoring of up to 8 input lines for event sensing or as a programmable divider.
5. Provide the period measurement of up to eight inputs.
6. Provide a frequency to count conversion of one input.
7. Provide for the control of a stepper motor having up to eight phases.
8. Provide a simplex asynchronous serial input.

9. Provide a simplex asynchronous serial output.

In addition to providing one of the above functions, the IDP can also handle simple parallel I/O through the unused port inputs or outputs.

III. FUNCTIONS OF THE INTELLIGENT DIGITAL CONTROLLER

The iSBC 569 Intelligent Digital Controller (IDC) is a versatile digital I/O processor. The IDC is designed to operate in a system using any one of the following three modes:

1. Intelligent Slave
2. Stand-alone System
3. Limited Bus Master

Additional power is obtained by the utilization of three OBS's to generate up to 48 parallel input/ output data lines.

In the intelligent slave mode, the controller's RAM is shared between the on-board 8085A and the Multibus users via a dual-port controller. Thus, a single bus master can control several intelligent slaves using the dual-port RAM as the major communications path. Switches are provided on the board to allow the user to reserve 1K bytes of RAM for use by the 569's processor only. This reserved memory is not accessible via the Multibus system interface and does not occupy any bus address space.

In the stand-alone mode, the entire system can consist of a single IDC, with cables, power supply and enclosure. An IDC can be installed at a remote site as a completely autonomous system.

The IDC may also be operated as a limited bus master when it is the only bus master in the system. Expansion memory and I/O boards may be connected to the IDC via the Multibus system bus to increase the input/output capabilities. This mode could be used to configure one IDC as a bus master with additional IDC's as intelligent slaves. This mode is not available with any other bus masters such as iSBC single board computers, disk controllers, or DMA devices.

Input/Output Functions

The I/O interface between the iSBC 569 Intelligent Digital Controller and the external devices to
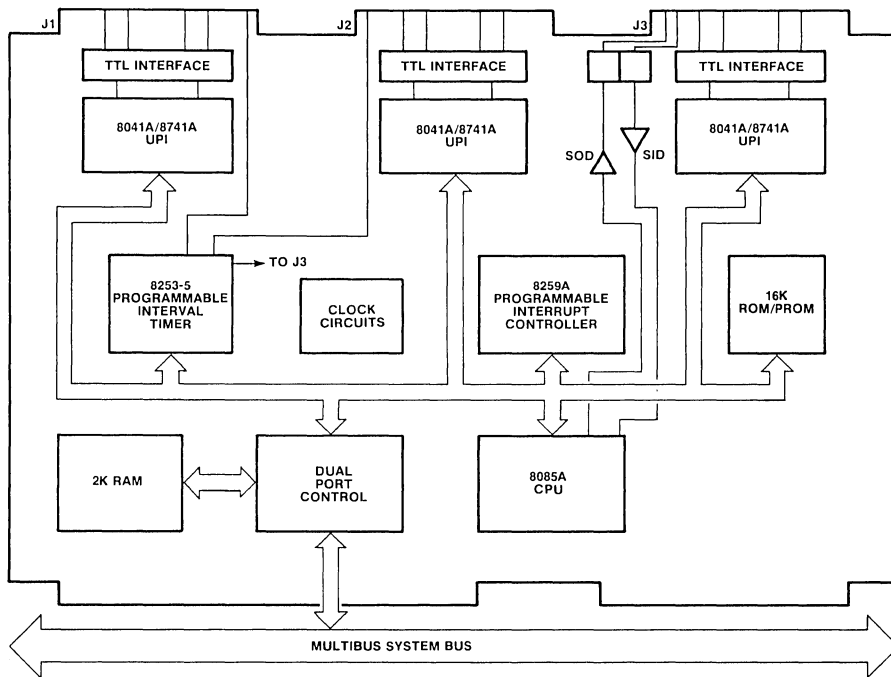
Figure 2. IDC Functional Block Diagram

which it is to be connected normally consists of various OBS devices. Each of these slaves has the ability to provide sixteen individual input and/or output lines. In addition, each provides two specialized input lines. The IDC is designed to accommodate up to three slave devices, so the normal I/O configuration of the board will consist of 48 digital data lines. If the specialized lines are considered, this number could be raised to 54. Sockets are provided for the insertion of drivers or terminators for use on the 48 digital lines. The 6 special purpose lines can only be used as inputs and are provided with pull-up resistors to terminate the input signals.

The driver/termination socket configuration limits the grouping of the I/O lines to be in groups of four. Any slave data line being used for an input must have its output latch placed into a logical 1 state so as to allow the input line to be controlled by the external signal.

IV. APPLICATION EXAMPLE

An example of the iSBC 569 controller in an application will help to explain the techniques used to implement a control system and to interface between the various functional units. The application chosen will consist of a typical use but will be simple enough to allow the design operations to be easily followed.

Suppose we choose to design a control system which will be produced as a subsystem to interface with and control a liquid applicator. As we go through the steps required to design and implement such a control system, we will see how the various hardware and software tools which are available from Intel can be utilized to allow easy implementation of the task.

Before proceeding, we will spend some time to insure there is a clear understanding about the definition of the liquid applicator. When this definition is complete, the design of the control subsystem can begin.

A liquid applicator consists of two functional parts: a device to control the flow of a solid material, and a device to control the flow of a liquid onto the material. We will actually be controlling two continuous process loops which are related by an input parameter which specifies the percentage of liquid to be applied to the dry material.

Figure 3 shows the components making up a typical weighbelt feeder. The operation of the feeder is straightforward. The vertical gate is adjusted manually to provide a desired gap between the conveyor belt and the lower portion of the gate. This will result in a nearly level distribution of material on the belt when it is moving. The weighbelt is connected to a load cell to provide information back to the control system giving the amount of weight on the belt at any instant. If we know the speed of the conveyor, it is simple to compute the amount of material flowing through the feeder during any time period. This

flow rate is known as the Mass Flow and is usually expressed as pounds per minute. The control of the feeder system can be provided by varying the belt speed until the desired flow rate has been obtained.

Our control system will be designed to control the belt speed and to monitor the weighbelt weight and any other parameters which we determine will be necessary to control the flow of material. A typical control process will require an optimum flow rate be established for each material of a different density. With a known material flow through the feeder, we can go about the process of applying the liquid flow to the material in order to complete our application example.

The second loop of the example will involve adding the liquid to the material coming from the feeder mechanism described above. Normally, the percentage of material to be applied is fixed by the
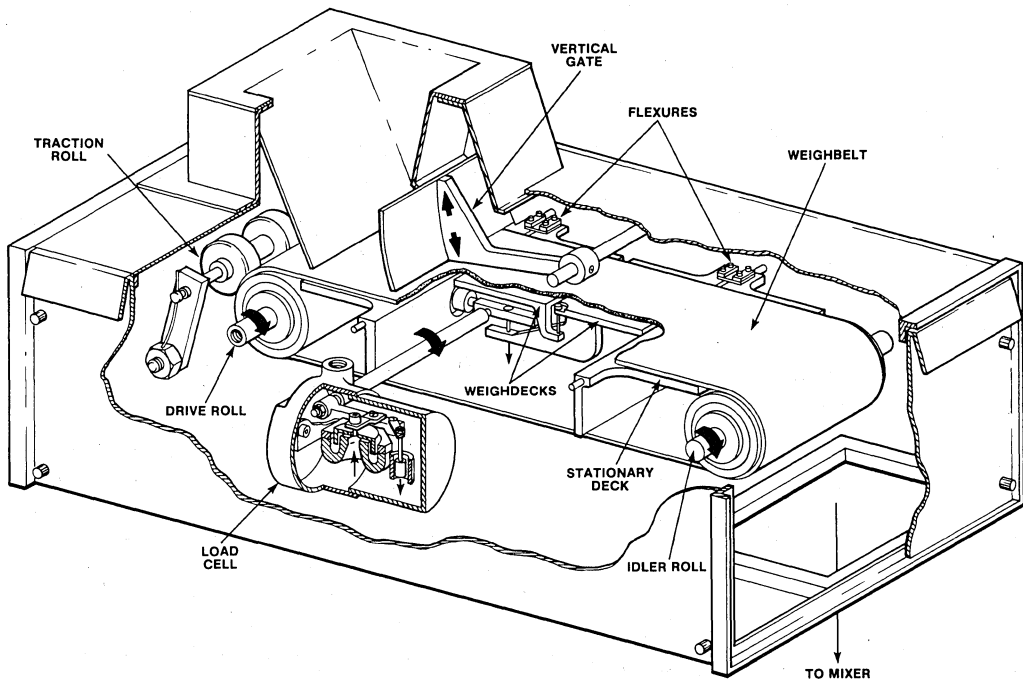
Figure 3. A Weighbelt Feeder

formula or mix design of the product which we are manufacturing. However, since the flow rate through the weighbelt feeder can and does vary (our first control loop will not always be able to exactly control the flow due to many conditions beyond our control), the liquid setpoint will constantly be changing as a function of the actual mass flow and the liquid percentage.

Figure 4 shows the liquid application piping diagram for the liquid portion of the control system. The items with which we will be directly concerned are the liquid flow meter and the control valve. The other components, while requiring consideration in an actual implementation, will be ignored in this aplication note for the sake of clarity. Let us consider the details of each control loop in more depth before we attempt to design the control system.

Mechanical Specifications

In subsequent portions involving development of the control system, we will be constantly referring to data regarding the mechanical specifications of the liquid applicator system. Therefore, we will establish a set of theoretical technical specifications for our system. Later, we will see how close the control system can come to providing a control which meets or exceeds these parameters. These specifications will be broken down into two sets of data, one for physical parameters over which we have no control, and a second for the desired control characteristics.

The physical data provides information on the mechanical design and will be used for guidelines in selecting interface equipment and in preparing software algorithms. The physical data is:

Operating Belt Speed —
1.1 to 180 feet per minute. Adjusted by a variable speed motor directly coupled to the belt pulley mechanism.

Feed Output Rates —
Adjustable over a 10:1 range with a maximum output of 960 pounds per minute.

Feeder Belt Characteristics —
The belt will be 9 inches wide by 2 feet in length when installed. The belt pulley rollers will have a radius of 4.5 inches.
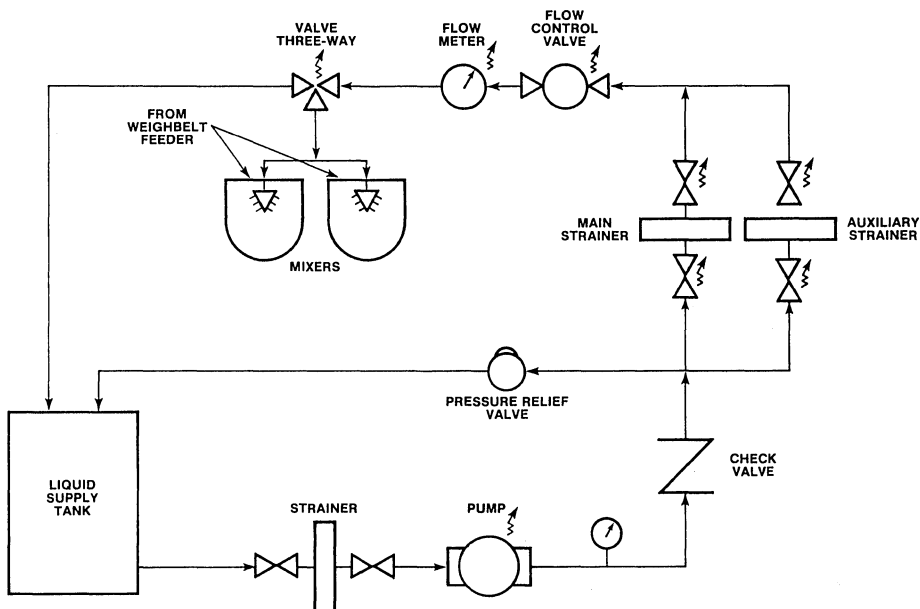
Figure 4. Liquid Flow Diagram

Feeder Weight Sensor —

The weighbelt feeder will incorporate a strain gauge load cell to measure the weight on the belt. Its linearity shall provide 0.1% of full scale range.

Liquid Flow Rates —

The liquid flow rates shall vary between 10.0 and 120.0 pounds per minute.

The desired operating characteristics of our control system will provide the following general responses:

Feeder Accuracy —

1% of full scale over a 10:1 range. The feeder will maintain the set feed rate within 1% of full scale over any one minute period. The minimum sample must be at least one pound.

Liquid Accuracy —

1% of full scale over the operating range. Must be able to track mass flow variations within the above limits.

These specifications will provide guidelines for the decisions which we will later make in providing a micro-computer control solution to the weighbelt feeder application.

Interface Requirements

A logical place to begin the consideration of the control system design is to examine the interface requirements and define the characteristics of the interfaces which will be required to implement the control. We will consider each element of the physical system separately.

Weighbelt Weight

The weighbelt weight will be sensed using a lever system connected to a load cell integral to the mechanical unit. The output of a strain gauge load cell is a low level (approximately 20 millivolts at full scale) analog output. Obviously, this signal must be somehow converted into a digital level before we can use its information to compute the actual mass flow across our weighbelt feeder. Our design process must define the characteristics of the digital signal so that the appropriate analog to digital converter system can be chosen. The design path can take any of several equally valid approaches, any of which will provide a functional control system. For the purposes of this application note, we will assume that the design path will utilize the Intel iSBC 569 Intelligent Digital Processor.

This assumption requires us to utilize only signals which can be generated or interpreted using the computer board and its associated OBS's. We will not be capable of handling an analog signal. Since some type of signal conditioning would be required of the low level analog voltage anyway, this does not impose any serious restrictions on our design. Indeed, it will cause us to consider a technique which provides excellent noise rejection characteristics. We will assume that a voltage to frequency converter (V/F) will be installed near the load cell and the frequency will then be transmited over a pair of wires to our digital interface. Commercially available converters provide a frequency output which varies between 0 and 10 kilohertz. With this in mind, we can continue with the development of the interfaces required in the application.

The load cell transducer will incorporate a local unit which generates a pulse train whose frequency is proportional to the weight upon the load cell. This mechanical arrangement is typical of many gravimetric feeder systems in use today.

For purposes of this application, it will be assumed that the system will be calibrated such that a weight of 10.00 pounds on the weighbelt will produce a pulse train frequency of 10 khz. No weight on the belt will generate a frequency of less than 30 hertz. The accuracy of the pulse output will be guaranteed to be proportional to the weight within 0.05%. Again, this is typical of devices available and in general use in similar applications.

The characteristics we have described above fall within the performance range of the iSBC 941 processor when operated in its frequency to count mode. If we assume a sample rate of 200 msec (this value should provide an adequate control characteristic since it is unlikely that the mechanical equipment can respond rapidly enough to warrant a faster control and sample time), the frequency count read by the iSBC 941 counter will range between 6 and 2000. System accuracy of reading the belt weight will thus exceed 0.1% of the full scale weight reading.

We will discuss the electrical and programming interfaces in subsequent sections of the application note.

Weighbelt Motor Control

The flow on the weighbelt will be controlled by changing the speed of the belt movement. Since the weighbelt is mechanically designed to maintain a constant bed level, the amount of material flowing will thus be adjusted.

The belt speed has traditionally been adjusted using either SCR controllers or by using variable transmissions between the motor and the conveyor belt. The increased utilization and development of stepper motors is leading toward greater use of direct stepper motor drives. This is the mode which will be utilized for this application.

The manufacturer's specifications for the weighbelt indicate that the following requirements exist for driving the device:

REQUIRED TORQUE — 149 LB-IN-IN
REQUIRED MAX SPEED — 2.54 REV/SEC.

Referring to typical manufacturer specification sheets for stepper motors, we find the torque vs. speed characteristics shown in Figure 5. Our application requires 2.54 revolutions/sec which translates to 508 steps per second when the stepper is used in a 1.8 degree per step mode. We can see that the requirements fall well within the capabilities of the particular motor.
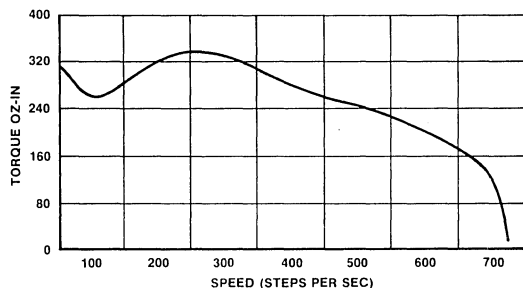
Figure 5. Stepper Motor Torque/Speed

At this point, we have four routes which may be pursued to actually interface with the motor. These are:

1. Utilize the iSBC 941 stepper mode to drive the stepper motor directly.
2. Utilize the iSBC 941 frequency generation mode to drive a standard stepper translator.
3. Utilize parallel outputs to provide a digital output to a stepper translater.
4. Utilize a 4-20 ma. current signal to a stepper translator.

Three of the above modes use a translator to drive the motor. If possible, we should strive to eliminate the cost of this intermediate device.

Again, we will refer to the published motor specification sheets. For our typical motor, the data is shown in Figure 6. The requirement for providing in excess of six amperes per winding exceeds the capabilities of the output drivers which can be installed on the iCS 930 termination board. We will be forced to either design a custom high power driver board or to use a translator module. To keep the application as simple as possible, we will choose the latter.

ELECTRICAL RATINGS 1.8 DEGREE STEPPING MOTOR

Motor Type	Time for One Step	DC Volts	Amperes Per Winding	Resistance Ohms	Inductance Millihenries
Ourtype	1.7 msec	2.3	6.1	0.37	2.4

Figure 6. Stepper Electrical Ratings

We have three choices left when the decision has been made to use a translator module. The use of a current output mode will necessitate the use of an external analog board. This is undesirable, both from the standpoint of interboard communication requirements, and from a cost effective basis.

The use of a parallel output would commit many of our output data ports and would require the installation of UPI modules or iSBC 941 modules to get the parallel output drivers. In addition, parallel digital input is not a common option of commercially available translators.

This leaves us with the use of a variable frequency output to provide stepping information to the translator module. This is a normal operational mode of the iSBC 941 processor and the required 508 hertz is within the normal output range of the device.

A definite advantage of our decision to use a stepper motor drive for the weighbelt is that we do not have to maintain accurate feedback and control algorithms to maintain the conveyor speed. Only a simple check need be made to verify that the conveyor has not stalled. The stepper motor will inherently maintain a speed proportional to the frequency rate.

The actual electrical and programming interfaces will be discussed in subsequent sections of this application note.

Weighbelt Speed Measurement

We have mentioned that a control system using a stepper motor for speed control can operate effectively in an open loop configuration. However, since a faulty component could result in failure of the motor to run, we must verify that the belt is indeed moving. This is easily accomplished by adding a magnetic sensor to the weighbelt rollers and counting the pulses generated as the device operates.

Typical magnetic sensors and ring magnets for installation on the weighbelt will provide us with ten pulses per revolution of a belt pulley. Since the pulley is operating at a maximum speed of 2.54 revolutions per second, we will receive between 0 and 25.4 counts per second. Using our sample period of 200 milliseconds, this means that we will count between 0 and 5 counts during each time interval. Our decision to use a stepper control loop rather than a conventional closed loop seems justified as we would obtain rather poor control with feedback having this poor of resolution.

We must make a decision to determine how the speed will be sensed by the control board. An obvious choice would be the use of an iSBC 941 processor operating in the period measurement mode. This would require using our third socket on the iSBC 569 host board and would leave us without the ability to use an additional device to support the liquid control loop. We should seek an alternative solution.

The iSBC 569 controller board provides an 8253 programmable interval timer. A first approach might be to attempt to configure one of these counters to provide an event counting mode and read the belt speed from the counter. However, this is not possible since we would be required to zero the counter after each reading and the counter does not load the preset count until a clock pulse is present. We would have no ability to distinguish between no belt motion and the belt motion which is the same as the previous reading!

An alternative approach is to create a software counter by routing the belt movement pulse to one of our interrupts and creating a program which will increment a counter. Each time a count is sensed, the software will increment a memory location by an increment which corresponds to the speed represented by one count.

Again, we will delay the discussion of the electrical and programming interfaces until subsequent sections of this application note.

Liquid Flow Control

The design of a control system to provide control of flow through a liquid valve is an integral part of the liquid pipe and plumbing design. To optimize the system operation and provide a system at the minimum cost, the integration of control and mechanical design must be made.

Several possibilities exist when making a decision as to which control valve to use in adjusting the liquid flow rate. The actual selection of the physical valve mechanism should be based upon the characteristics of the liquid flow. This decision is outside of the scope of this application note and will not be pursued. However, the valve actuator is a device which becomes an integral part of the control system and its selection is a function of the control system design.

Figure 7 shows the common control valve types which are used to vary the flow rate of liquids. The automatic control system we are designing precludes the use of a manual valve, so we must make our selection between the air actuated and the motorized control valve.

Classical control design has utilized air actuated valves almost exclusively. This type of actuator incorporates an intermediate transducer to

PROPORTIONAL CONTROL VALVES
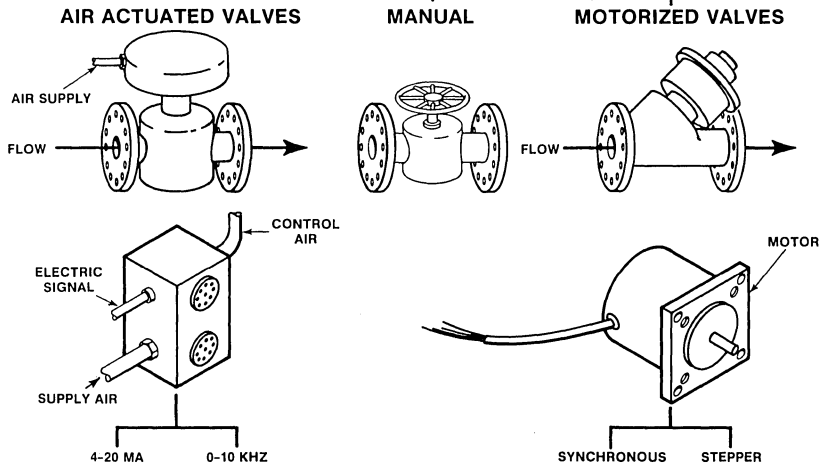
Figure 7. Control Valve Family

convert the signal generated by the control system into a variable air pressure. This air is used to drive a pneumatic control actuator. Two types of electrical to pneumatic transducers are in common use. The most prevalent converts a 4 to 20 milliampere control signal into a proportional air signal. The second type will accept a 0 to 10 khz pulse train and convert this to an air output.

Both of the above systems provide excellent electrical noise immunity and give reliable operation in industrial environments. They do, however, have disadvantages. A supply of air must be present at the control devices and this air must be maintained such that it is free from water and oil. In many cases, this presents costly installation and maintenance considerations. The use of computerized control systems has led to a recent concept of eliminating the intermediate conversion and using instead a digitally controlled actuator.

A stepper motor can be connected to the actuator

of the control valve to provide a simple and economical control path. The control outputs from the PID control loop can be sent to the iSBC 941 processor's command queue and the controller will handle the motor movements.

The electrical and programming interfaces of this interface will be fully discussed in subsequent sections.

Liquid Flow Measurement

The use of a liquid control valve to vary the liquid flow cannot in itself provide an accurate control loop. Because the flow rate through a fixed valve will vary with material densities, temperatures, and pressures, we must provide some type of feedback into our control algorithm. Thus, a flowmeter must be inserted into the liquid flow and its output returned to the system.

The control system designer can choose from several types of flow meters depending upon his requirements. Figure 8 shows many of the more
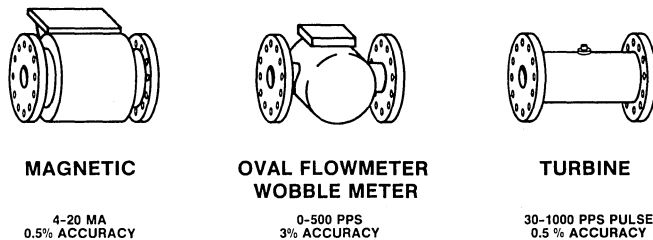
Figure 8. Flow Meter Classifications

3-73

standard classifications of flow meters. Our selection of the meter must take into account the type of electrical interface available from the meters. In our attempt to maintain a digital system which does not require additional support boards, we will reject the use of a magnetic flowmeter because this type of meter provides an analog type of output which would require the addition of another board into our control system. The wobble meter provides a digital pulse type output but its accuracy tends to discourage its use in a refined control loop. We will utilize the turbine meter for our liquid flow application.

The output of a turbine meter is a low voltage, low current AC signal whose frequency is proportional to the liquid flow rate. The manufacturers of the meters provide pre-amplifiers which convert the signal into 10 volt peak to peak square waves which are equivalent in frequenacy to the AC pulses. The operating frequency ranges typically from 100 to 1200 pulses per second.

It is desirable to measure the flow rate using a single iSBC 569 controller. If we consider that a 200 millisecond control interval will be used, the flow will result in a reading of between 20 and 240 pulses per sample period. These readings could be performed using an iSBC 941 processor, but we do not have the socket available for a fourth module, so we must consider utilizing another interrupt driven software counter as was done with the belt speed.

All control and monitoring equipment for our liquid control application has now been defined in such a manner as to be compatible with the utilization of a single iSBC 569 controller board. The actual interfaces to perform the interconnections and to provide control instructions can soon be considered.

Operator Interface

Finally, we must define the data communications which must take place between the controller, other system tasks, and the operator. Let us first consider the system control variables and the data which, if generated by the control process, might be useful to the remainder of the control system.

The first variable which comes to mind is the liquid flow setpoint. If we consider the entire control system, this parameter will be found to be actually expressed as a percentage of the total output material. For example, if we assume the recipe required the final product to consist of 5% liquid by weight, we would require that our control system add the correct amount of liquid to perform this task.

To allow maximum flexibility of the control system, we should allow selection of various density materials onto the weighbelt. A host processor with computational capabilities can calculate the optimum gravimetric feeder flow rate for the materials being combined.

The control system can provide an integration function to allow totalization of the amount of material which has been transferred through the system. A capability of outputting the amount of material which has passed over the weighbelt and the amount of liquid added will be included.

The implications of the parameter storage and generation will be dealt with later when the host/slave relationships of the iSBC 569 controller are discussed.

Interface Summary

We have defined the required interfaces which will be needed to perform our control task. These can be grouped into external and internal interfaces. The external interfaces are those which connect to physical pieces of external equipment.

These are summarized in Figure 9. The internal interface relates to the data which is to be passed between the iSBC 569 Intelligent Slave board and other boards which may be present on the MULTIBUS system bus. These data areas are shown in Figure 10.

V. HARDWARE CONFIGURATION

We have now defined the various components which we will utilize on the controller board to support the physical control and monitor hardware. Our next task is to provide an interface between the controllers and the equipment which we are to control. In so doing, we will define the hardware I/O assignments for the iSBC 941 processors and for the counters which we will be utilizing. The following paragraphs will deal with the optimization of this configuration.

	DEVICE	SIGNAL TYPE	BOARD ELEMENT
	WEIGHBELT MOTOR	10 VDC PULSE	iSBC 941
	WEIGHBELT WEIGHT	10 VDC PULSE	iSBC 941
	WEIGHBELT SPEED	110 VAC PULSE	8259A INTERRUPT
	LIQUID VALVE	5 VDC MULTIPHASE	iSBC 941
	LIQUID FLOW	10 VDC PULSE	8259A INTERRUPT

Figure 9. Control/Monitor Signals

INPUTS	OUTPUTS
GRAVIMETRIC FLOW	ACCUMULATED SOLIDS
LIQUID PERCENTAGE	ACCUMULATED LIQUID

Figure 10. Communication Signals

Controller Interface

Good design practice dictates that we should provide optical isolation between the controller and the external equipment when designing for an industrial environment. The optical isolation is included if we utilize the Intel iCS series of signal conditioning/termination boards. We find that we have two types of digital termination panels available, one for low current, low voltage applications and second for higher current and voltage uses. If we base our choice on the data provided by Figure 8, we will lean toward using the iCS 930 panel for our interface. This board can handle a mixture of signal levels and will support up to sixteen individual lines, providing almost double our needs.

Even a cursory glance at the iSBC 569 controller will provide the knowledge that three edge connectors are utilized to bring the OBS signals from the board. This would indicate that the simplest (and most costly) solution is to use three termination panels. Obviously, we should investigate further before making such a decision. Three possibilities are readily apparent. First, we might

perform some type of re-routing of data lines on the board so as to use only one connector. Second, we can use more than one connector on the ribbon cable and perform a parallel connection of the various lines and choose them so that no duplication of lines results. Finally, we can use some scheme of connecting three cables to the board and use the optional Port C connectors on the termination panel.

The schematic drawings of the IDC indicate that only six of the OBS I/O lines of each processor socket are broken by wire wrap jumper posts. All of the lines so configured are on the Port 2 data lines. Unless we decide to cut etch and add soldered wires, we will not be able to configure our board with this technique. Some further investigation is in order before we can make a decision. The use of a parallel output technique using multiple connectors on a single cable seems to present a feasible approach if we can work out an assignment of I/O which will not cause conflicts. We will begin by building a trial port assignment table in which we will assign the required functions to input/output ports. We will group the inputs and outputs into groups of four to handle the terminator/driver arrangement which is built into the board. This table is shown in Figure 11. We obviously have a small problem. We have

Port	Socket 1	Socket 2	Socket 3	Direction
10		Weight In		In
11				In
12				In
13				In
14				
15				
16				
17				
20	Conv. Mtr.			Out
21				Out
22				Out
23				Out
24			Valve Ph. 1	Out
25			Valve Ph. 2	Out
26			Valve Ph. 3	Out
27			Valve Ph. 4	Out

Figure 11. UPI™ Socket to Terminator Initial Assignments

not yet shown the signals from the conveyor speed and the liquid flow into the on-board interrupt counters. The schematics show that these signals are brought onto the board on the edge connectors but the locations correspond to Port C lines which do not exist on the iCS 930! We have available input lines on the Port 1 connectors but there is no provision to break the signal on the board to route it to the counter interrupts.

If we move on to the third alternative, we find that the interconnection paths caused by tieing various lines together cause even greater problems. Either some fact must have been overlooked, or we must consider the use of more than one terminator board.

Figure 11 indicates that three lines are available on the Port 2 data lines which go to jumper posts and which could be used if they were not part of an output driver of Port 20. If some technique can be found to use these "output" lines as inputs, our problem will be solved. The use of an open collector driver can provide us with the ability to use the line as an input so long as the drivers are turned off! This should be no problem as we can force the outputs to this state either through the appropriate jumpering of inputs or by outputting data to the OBS 1 ports corresponding to these bits. The resulting electrical configuration can be seen in Figure 12.
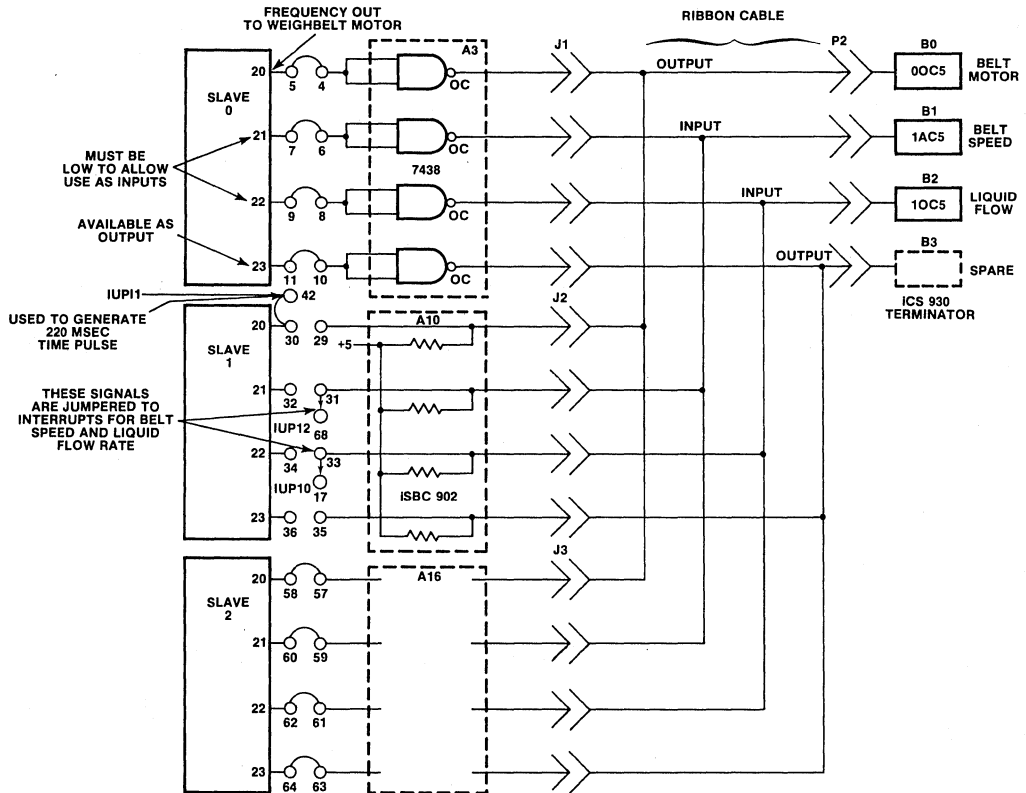
Figure 12. Port Assignments 20-23

Let us examine the implications of performing this interconnection. The physical layout of the board and the use of the terminator/driver sockets causes the I/O lines to be grouped into sets of four data lines. We must choose which of the three iSBC 941 modules will be responsible for supporting each of the lines. In Figure 12, we can see that the belt motor is driven by OBS Socket 1, Bit 20. This requirement has placed output drivers onto data Bits 21, 22 and 23. Our requirement is to provide two signals which can be routed to the counter inputs so we must place a terminator into either socket A10 or A16. We have arbitrarily chosen to use socket A10. The use of the terminators in parallel with the drivers will not create a problem so long as those lines which are used as inputs

have the driver in the high impedence state. This is done by requiring that the output Bits 21 and 22 of the device placed into socket 1 are driven low. Finally, we see that the remaining Bit 23 may be used as a general purpose output line if it becomes required.

The wiring configurations for the remaining connector groupings are shown in Figures 13, 14 and 15. In Figure 13, we see the assignments which can be used for Bits 10, 11, 12 and 13. We have earlier defined that an iSBC 941 processor would be used in a high speed frequency counting mode to determine the weighbelt weight. This device will be placed into socket 2. The use of this mode precludes the use of any general purpose
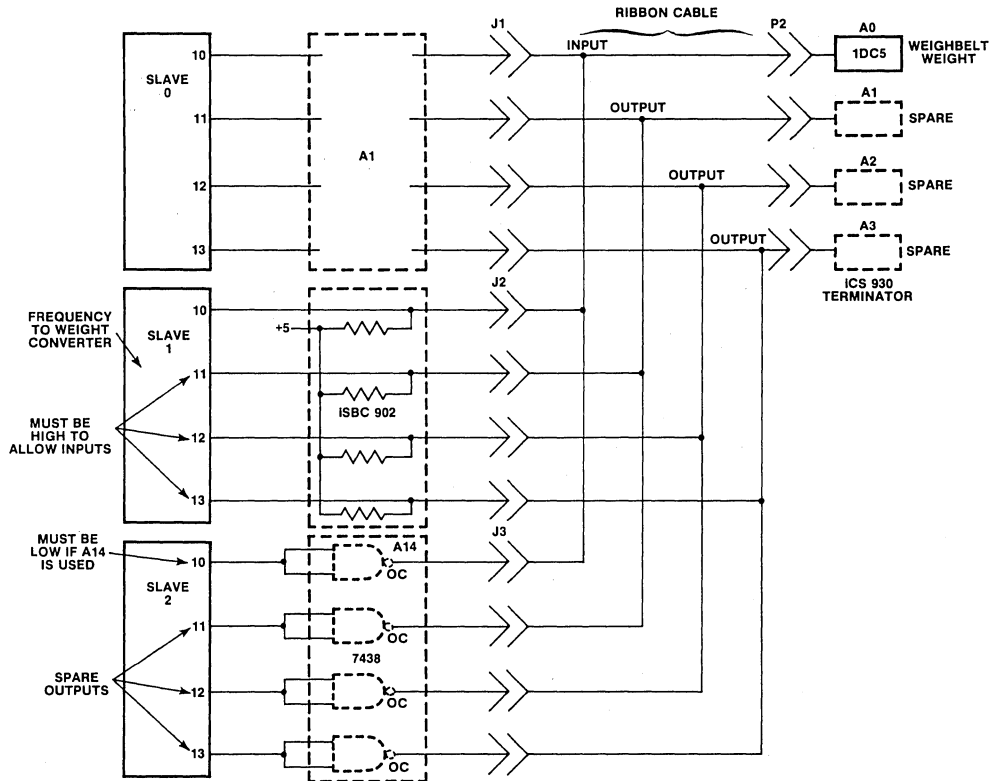
Figure 13. Port Assignments 10-13

input/output operations of the processor if we desire to maintain maximum accuracy of the frequency measurement. We will arbitrarily choose to use Bit 10 as the location of the frequency count input. This will necessitate installing a terminator into the socket corresponding to the processor input. If required, we can install open collector drivers into socket A14 and use the remaining three bits for general purpose outputs. If this is done, care must be taken to assure that Bit 10 of the device which is placed into socket 3 is placed into a low state as was done in the preceding example.

The interconnection scheme for Ports 14 through 17 can be seen in Figure 14. Note that no ports of this group are dedicated to our defined control functions. These four bits may be used as inputs or outputs as required by the application. For example, we have ignored the fact that actual control loops incorporate solenoids for flow control routing. The unused bits can be used to perform these tasks.

Figure 15 shows the interconnections for the remaining group of bits. There are several features shown on this drawing which should be discussed in some detail. Let us first consider the remaining function which we must implement. This is the control for the liquid valve stepper motor. An iSBC 941 IDP operating in the stepper mode will provide the necessary control functions to drive the motor. Since all four of this group's
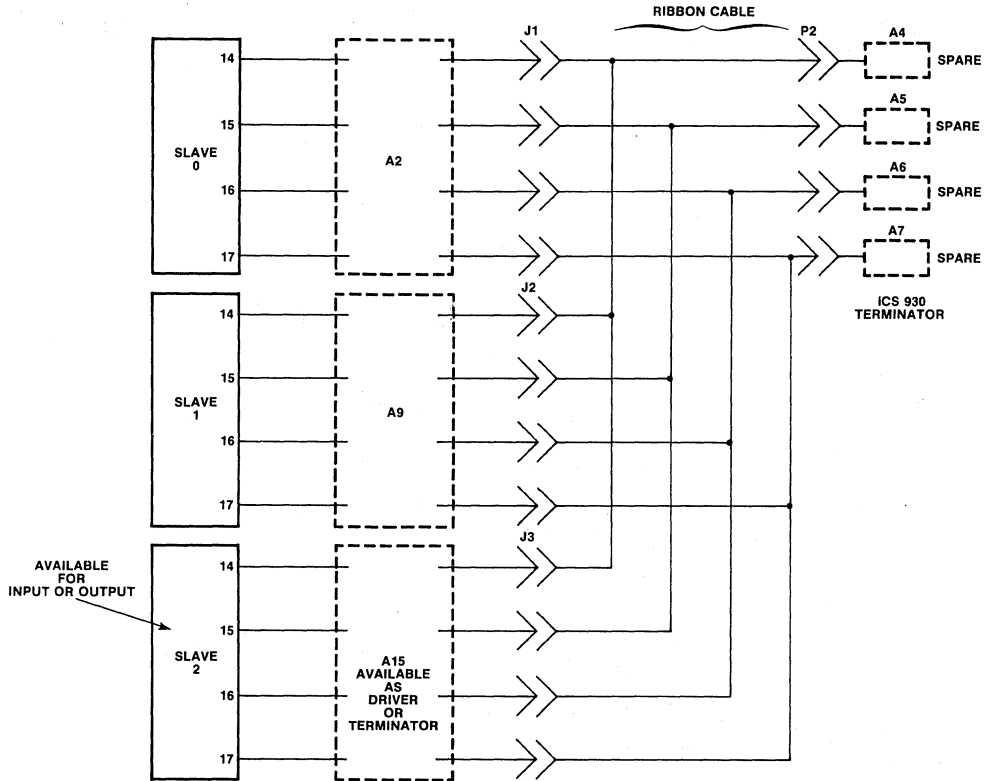
Figure 14. Port Assignments 14-17

data lines are committed to drive the four phases of the stepper motor, there are no other functions available.

An important feature of the iSBC 941 processor is illustrated in Figure 12. This is the ability to enable the processor to generate an interrupt at some point in its operation. We have earlier indicated that we will use the processor in socket 2 (the frequency counter) to provide us with a 200 msec time reference. When the iSBC 941 processor is enabled with an ENFLAG command and is operating in the frequency count mode, it will generate an interrupt on its output line, Port 25. Figure 15 shows how this interrupt can be connected to the host board's internal interrupt input structures.

The hardware configuration has been defined through Figure 14. The actual implementation can be handled through the use of the various wire-wrap jumpers on the IDC. Drivers and terminators can be installed as indicated in the preceding discussion.

VI. SOFTWARE CONFIGURATION

As with most computer controlled systems, the actual implementation of the task is handled with software. In older designs and in many mini-computer systems, this task has become formidable and has resulted in cost over-runs and schedule delays. Intel provides many tools for use by the designer to prevent this type of problem and to assist him in easily creating a workable and well
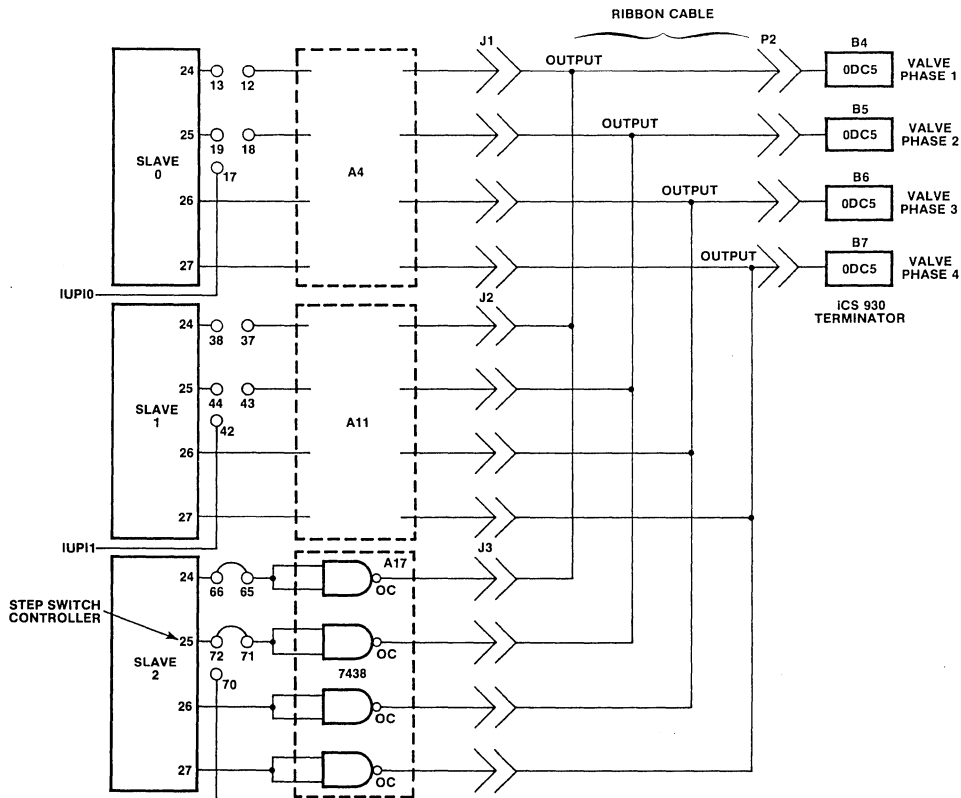
Figure 15. Port Assignments 24-27

documented software configuration. Let us look at some of these tools in more detail and consider how their use will help us to write our programs easily and quickly.

High Level Programming Languages

A valuable tool, which Intel provides the designer of small control systems, is the ability to program even the smallest systems using a high level programming language, PL/M-80. This language offers relatively efficient and structured, programming capabilities. It has been determined that PL/M-80 users can expect to use between 1.1 to slightly more than 2 times as much program memory as would be used for the same task written in assembly language. At the same time, the programmer's time to code a task will be considerably less than if he were to use assembly language.

The PL/M-80 Programming Manual indicates that the language is highly structured and lends itself very well to handle logical type operations. Its weakness in handling complex mathematical computations is compensated by the ability to combine the user application software with packaged Intel support software.

Fundamental Support Packages

The Intel 8080/8085 Fundamental Support Package (FSP) provides a package of application subroutines and functions which can be called from programs written in either assembly language, PL/M-80, or in FORTRAN-80. It uses a standard set of data structures and a unified status and error reporting scheme. Nine major groups of operations are fully supported by this package. These are:

1. A primitive fast string handling and integer arithmetic capability without error reporting.
2. A binary integer arithmetic package which performs operations on both signed and unsigned integers of various lengths in binary representation.
3. The floating-point arithmetic package which provides operations on floating point numbers in four formats: single precision, single-precision extended, double precision, and double-precision extended.
4. The decimal arithmetic routines which perform integer and fixed point computations on numbers which are stored as strings of ASCII characters.
5. A string handling section which contains routines to transform strings and to extract and insert substrings. A routine for scanning of general input and one for formatting of general output are included.
6. Routines for number conversion, for numeric I/O transformation of data from one format to another, input scanning of numeric strings, and formatting of numeric strings for output are also available.
7. The floating point transcendental function section provides trigonometric, exponential, and other transcendental functions.
8. The statistics routines compute the mean, variance, and standard deviation of one group of statistical data, and the covariance and correlation factor of two groups of data.
9. Finally, the PID procedures provide the user with a version of the classical Proportional, Integral, Derivative control algorithm.

Clearly, the use of the FSP support programs enhance the logical PL/M-80 program operations.

Host/Slave Relationship

Before we proceed with our development, we should take some time to examine the relationship between our iSBC 569 IDC and other controllers which may be installed in the system. The utilization of intelligent slave boards provides the capability to develop control concepts to an extremely high level if certain guidelines are followed. We will therefore assume that the control solution which we are developing will be but a part of an over all control concept which utilizes multiple controllers sharing common resources.

This concept allows us to develop control algorithms for each sub-process within our overall control system. This development can provide independent design and implementation of each process. A host processor can be used to provide any required inter-process communication tasks and to provide the operator interface. We have previously indicated that the operator interface will provide some means to adjust the weighbelt

feeder setpoints and the liquid ratio. It should also allow the operator to display the current status of the process. Since these operator interface functions are but a part of the overall control functions, the interface should be programmed such that maximum flexibility can be gained through its use. Fortunately, such an interface is available using Intel's RMX/80 BASIC-80.

RMX/80 BASIC-80 Interpreter

The RMX/80 BASIC-80 Interpreter is a high level language interpreter with extended disk capabilities. It operates on iSBC 80 Single Board Computers and allows the interpretation of BASIC-80 source code into an internally executable form. Many other features are available and many configurations are possible depending upon the exact system requirements (refer to the *BASIC-80 Reference Manual*, 9800758).

Maximum utilization of the operator interface with a minimum of development time can be achieved with the preconfigured version of the software/hardware package. This will provide us with complete disk I/O capabilities and the ability to easily program and maintain any programs which may become necessary to implement the interface. The actual implementation of the interface will be done later, after we have defined the control task.

Software Tasks

The task of preparing the software can be broken down into three major groupings or tasks. These are defined to be:

Prepare the Software Drivers.
This involves defining the relationships between the control algorithm parameters and the input/output hardware devices and creating software to implement these definitions.

Prepare the Control Algorithm.
This will involve developing a control algorithm which defines the relationships between the various system parameters. This algorithm will draw heavily upon the resources of the FSP programs and the software drivers which relate the parameters to the physical hardware.

Finally, the operator interface must be defined which will relate the parameters used in the control scheme to other controllers and to the operator. This will allow the control task to interact in such a manner as to provide a meaningful element of the overall control concept.

VII. SOFTWARE DRIVERS

Before developing the actual control algorithm, we must create the drivers which communicate with the three iSBC 941 processors in their assigned operating modes. We will define two driver sections for each processor, one to handle the initialization, and a second to provide the ongoing communications as required by the control algorithm program.

Motor Speed Control Processor

The first processor which we will discuss is to be located in slave socket number 0 and will be used to produce a variable frequency output. Let us consider in some detail how this can be accomplished using an iSBC 941 Processor. First, consider the task of initializing the device to the primary function operating mode, FREQ.

Referring to the *iSBC 941 Industrial Digital Processor User's Guide,* we find that the initialization requires the sequence of commands and data shown in Figure 16. We will identify the meaning of each of these terms and create a software

Description	Command/Data
Request INIT	C
FREQ Select	D
Scale Factor	D
Output Enable	D
Initial State	D
P20 Delay	D
P20 Period	D
Request PAUSE	C

Figure 16. FREQ Initialization

program which will handle the required initialization of the processor. The purpose and use of the various commands to the processor are well defined in the user's guide and will not be repeated here.

The first byte of data, which must be sent following the initialization command, is the data byte signifying that the operational mode is to be the frequency output. This is defined in the manual as being equal to the data byte "0B5H" or "035H" as expressed in the hexadecimal numbering system. The choice of values to be sent is dependent upon our desire to utilize the internal or external time reference period for the operations. If we utilize the internal time reference, our minimum increment or resolution of operations will be 86.72 microseconds.

To determine if this speed is adequate for our frequency generator, we must consider the impact that this resolution has on the output. A 550 hertz signal has a period of 1.82 milliseconds. If we increase this period by the 86.72 microsecond time reference, we find that the next increment in the frequency generators output will be approximately 372 hertz. This resolution is certainly not adequate to meet the motor control requirements! We should consider using the external clock to provide the time reference. One of the 8253 Interval Timers on the iSBC 569 board can be used to generate a reference time. If we arbitrarily choose to use a 10 microsecond reference to the IDP, we find that the worst case resolution for the 550 hertz signal becomes about 4 hertz. This is certainly within our requirements of motor control. The primary function signal should then be sent as a "0B5H".

The second byte is used to establish a scale factor for the processor. This scale factor is used to generate the basic time increment which can be used to establish the frequency output; that is, the minimum time increment which can be used to establish a period or pulse width will be the scale factor times the reference time period.

In our case, because of the wide frequency output range, we cannot specify the scale factor at initialization (later data will show the need for multiple scale factor ranges). We will then only need to send some arbitrary value at initialization to allow the processor to complete its initialization sequence.

The Output Enable data byte is used to select which of the Port 2 output bits are to be used to generate the output signals. The hardware configuration established earlier placed the output onto Bit 0 of the port, so this data byte shall be specified as a byte having only Bit 0 set to a logical one or equal to 01H.

The Initial Output parameter specifies whether each bit selected as an output by the output enable byte is to be initially set to a logical one or zero when the processor is first enabled. For this application, it really does not matter, but we will arbitrarily pick the state to be equal to zero. The byte will be defined as being set to 00H.

The Delay parameter is used to define the waveform which will be generated and specifies the number of time increments which must elapse before the waveform will change states. Rather than to constantly vary the delay to maintain a square wave output, we can choose an arbitrary value of one time increment before changing state. The output will have a varying duty cycle as the frequency changes. This should cause no problems for the translator driving the weighbelt motor. The byte will be defined as being set to a value of 01H.

Finally, the Period of the waveform must be chosen. Again, this parameter will be changed according to the desired frequency, so only an arbitrary value need be sent. Indeed, since this is the last parameter, the value could be omitted entirely by sending the PAUSE command in its place.

The initial data definition can be defined using PL/M-80 language conventions as a block of six bytes as shown in Figure 17.

The actual communications between the host processor on the iSBC 569 board and the IDP utilizes the protocol explained in previous sections of this note. The status register of the IDP will be tested for the bit signifying that the input buffer

```
                    /* DECLARATION OF iSBC 941 #0 INITIALIZATION DATA */
        22   1          DECLARE FREQ                LITERALLY '0B5H';
        23   1          DECLARE SF                  LITERALLY '000H';
        24   1          DECLARE OUTPUT$ENABLE0 LITERALLY '001H';
        25   1          DECLARE INITIAL$STATE    LITERALLY '000H';
        26   1          DECLARE DELAY              LITERALLY '001H';
        27   1          DECLARE PERIOD            LITERALLY '000H';



                    /* DECLARATION OF iSBC 941 PRIMARY DATA */
        34   1          DECLARE INIT$0$TABLE(6) BYTE DATA (
                            FREQ,
                            SF,
                            OUTPUT$ENABLE0,
                            INITIAL$STATE,
                            DELAY,
                            PERIOD     );
```

Figure 17. Initial FREQ Data Field

full is not set. This will indicate that the device is ready to accept either a command or a data byte. The command to request a primary function will be sent. At this point, the processor will be expecting a series of data bytes as specified by the function being selected. A "Do Loop" can be used to effectively transmit this data to the device. The program to perform this function is illustrated in Figure 18.

```
                    /* REQUEST PRIMARY FUNCTION */
        44   2          DO WHILE ( (INPUT (UPI$0$STATUS) AND IBF) < > 0);
        45   3          END;
        46   2          OUTPUT (UPI$0$COMMAND) = INITPF;

                    /* LOAD INITIAL PARAMETERS */
        47   2          DO I=0 TO 5;
        48   3              DO WHILE ( (INPUT (UPI$0$STATUS) AND IBF) < > 0);
        49   4              END;
        50   3              OUTPUT (UPI$0$DATA)=INIT$0$TABLE(I);
        51   3          END;

                    /* TERMINATE PARAMETER LOADING */
        52   2          DO WHILE ( (INPUT (UPI$0$STATUS) AND IBF) < > 0);
        53   3          END;
        54   2          OUTPUT (UPI$0$COMMAND)=PAUSE;

                    /* START FREQUENCY FUNCTION */
        55   2          DO WHILE ( (INPUT UPI$0$STATUS) AND IBF) < > 0);
        56   3          END;
        57   2          OUTPUT (UPI$0$COMMAND)=LOOP;
```

Figure 18. IDP Initialization

When all required data parameters have been sent, the data portion of the initialization is terminated by sending a PAUSE command as shown in Figure 18. Note how, in each case before data or a command is sent, we wait until the input buffer is empty. Finally, the initialization is completed when we have sent the LOOP command. The processor will now be generating an output frequency as specified by the parameters.

Remember that, according to our earlier discussion and as we have shown in Figure 12, the unused output ports should be set to a logical low condition to allow the use of those lines as inputs to carry additional data into the controller. This should be done as a part of the initialization process. The secondary utility command, CLRP2 is used for this purpose. This process is illustrated in Figure 19.

We should next direct our attention to establishing a software interface which will take the desired weighbelt speed term and convert it to a frequency output suitable to drive the motor translator. We know that this will involve selecting a particular scale factor and period term which will generate the correct waveform. Previously, we established that, for a maximum frequency of 550 hertz, we need to establish a period of 1.82 milliseconds. Many combinations of Scale Factor and Period parameter will generate this time interval. Ideally, the smallest increment of change can be established by setting a constant period and modifying the scale factor. If we make some calculations, we will find that the fact that the scale factor is a byte value (giving us a range of between 0 and 255) limits the frequency range which can be produced using any one value for a period. It seems that we will be forced to vary both the period and the scale factor as a function of the desired frequency.

In Figure 20, we have plotted the frequency output for various values of Scale Factor and Period. Our

```
                        /* SET UNUSED BITS TO ALLOW EXPANSION */

        59   2          DO WHILE ( (INPUT UPI$0$STATUS) AND IBF) < > 0);
        59   3          END;
        60   2          OUTPUT (UPI$0$COMMAND)=CLRP2;

        61   2          DO WHILE ( (INPUT (UPI$0$STATUS) AND IBF) < > 0);
        62   3          END
        63   2          OUTPUT (UPI$0$DATA)=INITIAL$OUTPUT;
```

Figure 19. Secondary Utility Command
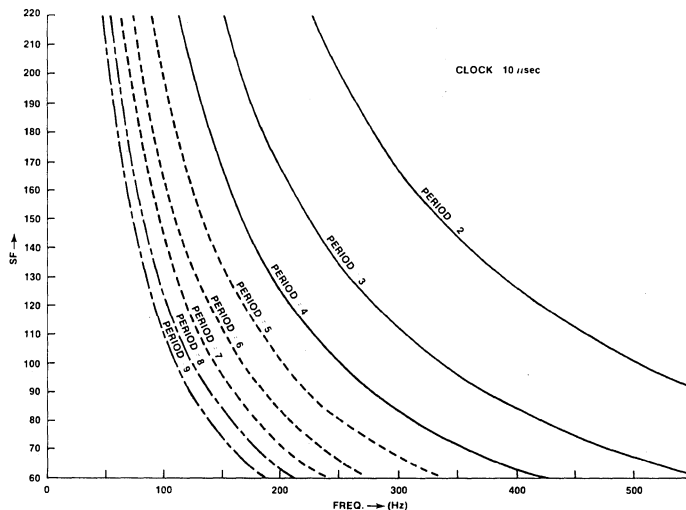
Figure 20. Frequency Vs. Parameters

intent is to maintain the highest resolution possible for the desired output range of 50 to 550 hertz. Choosing four period base parameters will provide us with acceptable waveform generation characteristics. We will choose the data sets of Figure 21 based upon the data shown in Figure 20.

The Period can be determined by examining the desired frequency range. The scale factor can be calculated from the equation:

SF = 10,000 / ((FREQUENCY) x (PERIOD))

Again, the PL/M-80 language program to implement the interface between the host and the IDP is easily constructed. For example, Figure 22 provides the code which will be required to determine the appropriate Period parameter and also illustrates the use of FSP programs to handle the mathematical calculations required to determine the corresponding scale factor.

The principles above can be expanded into a complete interface package to offload the host processor of the need to generate the frequency waveform to the translator of the weighbelt motor. The complete program for the processor can be found in Appendix A.

Weight Input Processor

The second use of an iSBC 941 Processor is to provide the capability of converting the high frequency inputs from the weight sensor of the weighbelt into a digital value equivalent to the actual weight on the belt. This frequency to digital conversion can be easily accomplished by the use of the Primary Function, FCOUNT.

Frequency	Period	Scale Factor	Resolution
. 50 to 165 Hz.	9	221 to 67	3 Hz.
166 to 225 Hz	5	121 to 89	3 Hz.
226 to 285 Hz.	3	147 to 117	3 Hz.
286 to 550 Hz.	2	175 to 91	6 Hz.

Figure 21. FREQ Output Ranges

```
                        /* COMPUTATION OF FREQUENCY RANGE */
    57    3                 IF FREQ < 285
                            THEN DO;
    59    4                    IF FREQ  < 226
                               THEN DO;
    61    5                       IF FREQ < 166
                                  THEN RANGE = 9;
    63    5                       ELSE RANGE = 5;
    64    5                    END;
    65    4                    ELSE RANGE = 3;

    66    4                 END;
    67    3                 ELSE RANGE = 2;

                        /* LOAD MATH ACCUMULATOR WITH 100,000 */
    68    3                 CALL MQULD4 (.IR,.HUNDRED$K);

                        /* TEST FOR MOTOR SHUTDOWN */
    69    3                 IF FREQ >1
                            THEN DO;

                        /* DIVIDE BY FREQUENCY */
    71    4                    CALL MQUDV2 (.IR,.FREQ);

                        /* DIVIDE BY RANGE FACTOR */
    72    4                    CALL MQUDV1 (.IR,.RANGE);

                        /* GET TWO'S COMPLEMENT FOR iSBC 941 SCALE FACTOR */
    73    4                    CALL MQUST1 (.IR,.FREQA);
    74    4                    FREQA=NOT (FREQA + 1);
    75    4                 END;
```

Figure 22. Period and Scale Factor Computations

The FCOUNT Primary Function is selected by sending the INITPF command followed by four parameters. The process is identical to that which was used in the previous example when we established the FREQ function. In this case, the sequence is described in the manual as is shown in Figure 23.

Description	Command/Data
Request INIT	C
Select FCOUNT	D
Input Select	D
Output Enable	D
Sampling Interval	D
Request PAUSE	C

Figure 23. FCOUNT Initialization

Let us examine the derivation of the terms which must make up the data table which will be transmitted to the processor in order to initialize it. The FCOUNT function does not allow the use of an external clock so we have no option as to which command will be sent to select this function. It is defined to be equal to 33H. This becomes the first element of the byte array used to contain the initial data.

The Input Select parameter describes which of the Port 1 inputs are to be measured. If we refer to Figure 13, we can see that a hardware assignment of Port 10 has been made for this function. This assignment corresponds to bit 0 of the parameter being set to a value of 1. The byte value for this parameter then becomes 01H.

The Output Enable byte is used to enable an output port corresponding with the input to change states when the Sampling Interval time has elapsed. Our system has a requirement to operate the control algorithm once each 200 milliseconds and we have previously indicated that the frequency counter would be used to establish this time interval. If the output is enabled and connected to an interrupt line, it will provide our system with the required pacer clock. The output bit from Port 20 will then be enabled to provide the interrupt. The parameter for this byte will be set to the same value as the Input Select and becomes 01H.

The Sampling Interval will establish the time interval to be used when sampling the input frequency. This time interval should be set to 200

milliseconds for our application. The parameter is then calculated from the equation:

$$\text{INTERVAL} = (\text{SAMPLE PERIOD}) / (0.02222)$$
$$\text{OR}$$
$$\text{INTERVAL} = (0.200) / (0.02222) = 9$$

The correct sampling interval for our control system should be set to a value of 09H.

A similar procedure can be used to send this data to the processor. The actual code used to implement the system can be found in Appendix A. Note that the unused bits of the device have been set to a predetermined value as was indicated by our hardware design of Figure 13.

Once the processor has been initiated and is performing its function, we need only wait until the device signals us that the 200 millisecond time interval has passed and that it is ready with the belt weight. When this interrupt occurs, we will read the data and perform our control functions. An interface must be established between the control algorithm and the processor which enables it to receive a value which represents the actual weight.

The total count received by the processor is available as a sixteen bit count made up of two eight bit bytes. The use of the Secondary Utility Commands, Read FCOUNT Measurements (RDFC0-RDFCF) allow the two bytes to be transferred into the host processor. We are using the first counter so we will use the corresponding commands, RDFC0 and RDFC1. An example of the procedure to read one of the count bytes can be seen in Figure 24.

The counter can be commanded to begin its next sample period by issuing a LOOP command to the processor. The two data bytes can be combined to form a 16-bit word and the resultant value divided by 2 to form a weight value. The division by two to obtain weight is required since the count range from 0 to 2000 corresponds to a weight of between 0 and 10.00 pounds; thus, each count has a value of 0.005 pounds. The integer numbers used in the control algorithm are fixed point with an implied scale factor of 100. The division by two provides a result which meets the criteria.

```
              /* GET INPUT COUNT LOW BYTE */
106   2        DO WHILE ( (INPUT (UPI$1$STATUS) AND IBF) < > 0);
107   3        END;
108   2        OUTPUT (UPI$1$COMMAND) = RDFC0;

109   2        DO WHILE ( (INPUT$1$STATUS) AND OBF) = 0);
110   3        END;
111   2        LCOUNT = INPUT (UPI$1$DATA);
```

Figure 24. FCOUNT Read Procedure

Appendix A provides the complete listing of the code which was used to interface with the processor assigned to the primary function, FCOUNT.

Stepper Motor Control Processor

The third example of utilizing the iSBC 941 Processor in an industrial application is provided by the processor installed into OBS socket 2. This device is used to drive a stepper motor which, in turn, controls the liquid valve position. Again, we will break the discussion into an initialization and an interface operational mode.

We find that the User's Guide indicates that initialization to the STEPPER Primary Function is performed by sending the INIT command followed by up to 21 data bytes. Figure 25 provides the table which shows the necessary parameters for this mode.

The technique used to place the processor into the desired function is the same as we have seen with the two other processors so we will not spend time dealing with the communications sequence. Instead, we will examine the techniques which can be used to determine the values of the initialization parameter bytes.

STEPPER is requested by sending a data byte of either 17H or 97H following the INIT command. Remember that the significance of setting bit 7 of the data high is to request that an external clock be used by the processor. There is no reason to use an external clock for our application, so we can choose a function request byte of 17H.

The remainder of the data is used to define the waveforms which are necessary to drive the stepper motor. We will derive the values for these parameters by beginning with the manufacturer's data sheet and moving until we have determined the correct value for each byte of data.

The motor chosen for this application utilizes four phases to drive the shaft. The data sheet provided

Description	Command/Data
Request INIT	C
Select STEPPER	D
Select Scale Factor	D
Output Enable	D
Output Polarity	D
Common Period	D
P20TRAN1	D
P20TRAN2	D
P21TRAN1	D
P21TRAN2	D
P22TRAN1	D
P22TRAN2	D
P23TRAN1	D
P23TRAN2	D
P24TRAN1	D
P24TRAN2	D
P25TRAN1	D
P25TRAN2	D
P26TRAN1	D
P26TRAN2	D
P27TRAN1	D
P27TRAN2	D
Request PAUSE	C

Figure 25. STEPPER Function Initialization

information for both a Four-Step Input Sequence (1.8 degrees per step) and for an Eight-Step Input Sequence (0.9 degrees per step). We will use the 1.8 degree step angles for our example and application. The data provided by the manufacturer is shown in Figure 26. The first task is to convert the switch state diagram into a desired waveform for each of the four phases. This has been done in Figure 27.

Beginning with Scale Factor, let us determine the required data parameters which will yield a stepper controller compatible with our motor. The Scale Factor will provide the minimum time period for one step to take place. The minimum time which we can specify is a function of both the motor characteristics and of the TRP for the primary function, STEPPER. The minimum TRP is determined by referencing the IDP User's Guide for the desired function. In this case, it is found to be $325 + (13 \times B)$ where B is the number of phases
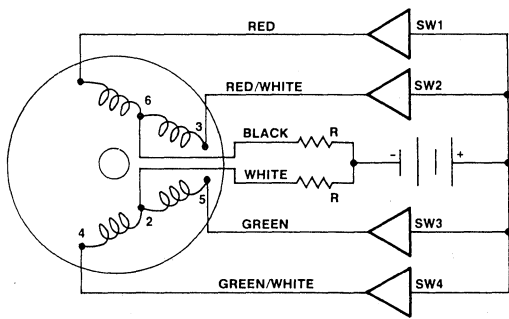
DC STEPPING CIRCUIT

EIGHT-STEP INPUT SEQUENCE

STEP	SW1	SW2	SW3	SW4
1	ON	OFF	ON	OFF
2	ON	OFF	OFF	OFF
3	ON	OFF	OFF	ON
4	OFF	OFF	OFF	ON
5	OFF	ON	OFF	ON
6	OFF	ON	OFF	OFF
7	OFF	ON	ON	OFF
8	OFF	OFF	ON	OFF
1	ON	OFF	ON	OFF

FOUR-STEP INPUT SEQUENCE

STEP	SW1	SW2	SW3	SW4
1	ON	OFF	ON	OFF
2	ON	OFF	OFF	ON
3	OFF	ON	OFF	ON
4	OFF	ON	ON	OFF
5	ON	OFF	ON	OFF

Figure 26. STEPPER Motor Input Sequence
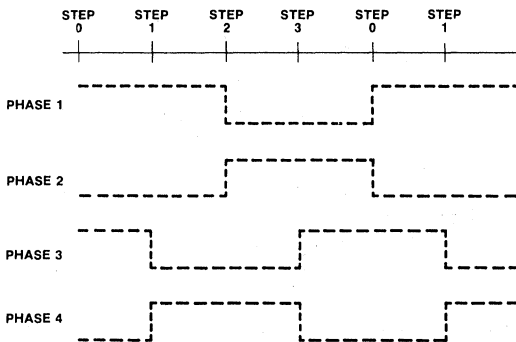
Figure 27. STEPPER Motor Waveforms

which are used. The result will be expressed in terms of processor cycles and can be converted into time by multiplying by 2.71 microseconds per cycle. This works out to be:

$$325 + (13 \times 4) = 377 \text{ PROCESSOR CYCLES}$$
$$\text{OR}$$
$$377 \times 2.71 = 1.021 \text{ MILLISECONDS}$$

Now, let's examine the minimum time which can be utilized by the stepper motor. This is given in the manufactuer's data sheets as being 2.86 milliseconds for the motor which we have chosen to

use. This value must be used to compute the Scale Factor for this application. The Scale Factor is computed by dividing the minimum step time by 86.72 microseconds or:

SF=2.86 MILLISECONDS/86.72 MICROSECONDS=33

This number is entered into the processor using two's complement which becomes equal to 0DFH.

The Output Enable is used to specify which of the eight possible control outputs are to be used to control the motor phases. The motor phase assignments to I/O ports was made in Figure 15 and indicates that Ports 24 through 27 will be enabled for the primary function. Setting the corresponding bits provides a parameter to be sent to the processor of 0F0H.

The rest of the parameters deal with providing a definition of the waveforms generated in Figure 26 to the processor. The following paragraphs deal with the operations required to convert the graphic representation into data parameters.

Each phase must be initialized to an initial output state which corresponds to the signal level shown for Step 0 of Figure 27. A "1" will be placed into the bit corresponding to each of the port's output bits which are to be in a logical one state upon

reaching step 0. We see that Bits 24 and 26 are set corresponding to phase 1 and 3. The data byte for Initial Output is thus defined to be 050H.

The Period parameter for a stepper motor function corresponds to the number of steps which are defined in the motor's step sequence. Our example uses a four step sequence so the Common Period will be set to a value of 04H.

The remainder of the initialization parameters define the transitions of each of the phases. This involves the examination of the waveform and noting the points at which the output level changes. This data can be input to allow the device to accurately produce the control waveforms for any stepper motor control mode. We are not using the first four output bits so the transition definitions for these outputs is meaningless and will be output as zeroes. The waveform for output Port 24 shows a transition at steps 1 and 3. The parameter for the first transition of Port 24, P24TRAN1 is defined to be 00H. Likewise, the second transition, P24TRAN2 is set to a value of 02H.

The technique used above can be continued to define the constants, P25TRAN1 and P25TRAN2 as being the same as for Port 24 or 00H and 02H respectively.

The transitions for the phases driven from Port 26 and 27 can be seen to occur at steps 1 and 3 so the data for those parameters can easily be seen to be set to 01H and 03H for each port.

The initialization table can be sent to the processor using the same techniques as were used for the processors discussed previously. The complete program for the initialization can be found in Appendix A.

A driver must next be prepared which will be used to provide the interface between the control algorithm and the IDP processor which supports the stepper motor. When the STEPPER primary function is used, a queue is utilized for supporting the step commands to the motor. Each command to the stepper consists of a data byte signifying the step rate to be used and a data byte which provides the signed magnitude of the number of steps to be moved. Using the motor to control a flow control valve allows us to use a constant step rate, but some type of program must be prepared which will convert the signed two's complement representation of the position from the control algorithm to a signed magnitude format.

The number conversion is easily done and the PL/M-80 programming code to perform the format change is shown in Figure 28.

The data queue allows up to six movement commands to be present and waiting to be serviced by the IDP. If the processor is behind in its operations and cannot accept a seventh request, the host must wait until one of the requests in the queue has been serviced. The queue status bits can be tested to determine if room exists for another command and the "queue not empty" bit can be tested to verify that all requested movements have been completed. Normal operation of our motor should be such that the queue is not allowed to fill to its maximum capacity.

```
                          /* SUPPORT CONVERSION TO SIGNED MAGNITUDE NUMBER */
            141    3            IF POSITION > 127
                               THEN DO;

                          /* GET MAGNITUDE OF MOVEMENT */
            143    4                POSITION = 256 - POSITION;

                          /* SET SIGN FOR CCW ROTATION */
            144    4                POSITION = POSITION OR REVERSE;
            145    4            END;
```

Figure 28. Number Format Conversion

The code which is required to test the queue and to send a stepper movement request is shown in Figure 29. The complete code can be seen in Appendix A.

VIII. APPLICATION SOFTWARE

Having developed the software which is required to support the Industrial Digital Processors, we can now devote our time to the task of implementing the application software and of handling any programs which are required to support functions unique to the host iSBC 569 board. This software can be grouped into two general categories, initialization programs, and control algorithm programs.

Initialization Programs

The initialization of the iSBC 569 involves setting up the required configuration of interrupt handling and of the devices which are installed into the slave sockets. For the purposes of this application, we will include some system diagnostic capabilities within the process. These routines will be executed each time a RESET or a POWER-UP occurs. Only the highlights of the code used will be presented in detail; however, the complete listings of the initialization programs can be found in Appendix A by referring to the BCKGND Program listing.

A unique feature of using the iSBC 941 processors is their ability to provide, upon request, an identification code. The initiation diagnostic program takes advantage of this fact by interrogating each processor and verifying that the correct ID code is returned. If any of the processors have failed catastrophically or if the internal data bus of the host board has failed, the program will provide an indication of this fact.

Each of the slave processors has, associated with it, an individual hardware reset line which is under the control of the host. A reset or power up condition will cause the control lines to reset to the state which hold each slave in a reset state. Before any slave can be used, it's associated reset line must be de-activated. This is done by sending a logical one to the corresponding bit of the Reset Latch. Other bits of the Reset Latch can be used to illuminate the on-board LED or to generate an interrupt to another board on the Multibus data bus.

A special PL/M-80 command is utilized to disable the reset interrupts of the 8085A host processor. Execution of this command will allow all servicable interrupts to enter via the 8259A Interrupt Controller. The command which will mask off the unused interrupt structure is shown in Figure 30.

The initialization process must also initialize the FSP Integer Record. This will allow the use of the math support routines which will be required to support the control algorithm.

```
                              /* VERIFY THAT QUEUE SPACE IS AVAILABLE */
            146    3              DO WHILE ( (INPUT (UPI$2$STATUS) AND QF) < > 0);
            147    4              END;

                              /* REQUEST DESIRED STEP RATE */
            148    3              DO WHILE ( (INPUT (UPI$2$STATUS) AND IBF) < > 0);
            149    4              END;
            150    3              OUTPUT (UPI$2$DATA) = STEP$RATE;

                              /* REQUEST STEPPER MOVEMENT */
            151    3              DO WHILE ( (INPUT (UPI$2$STATUS) AND IBF) < > 0);
            152    4              END;
            153    3              OUTPUT (UPI$DATA) = POSITION;
```

Figure 29. STEPPER Movement Request

```
                              /* MASK OUT THE RESET INTERRUPTS OF THE PROCESSOR */
            34     1              CALL S$MASK (MASKS);
```

Figure 30. PL/M-80 Sim Instruction

Control Algorithm Programs

The program which actually handles the control algorithm for the two loops can be found in Appendix A, MAIN$CONTROL. The flow of the program is straightforward and can easily be followed by reading the listing. The operations are primarily handled by the use of repeated calls to the FSP integer math routines and to the processor interface modules which we have previously generated.

It is beyond the scope of this application note to dwell upon the techniques which were used to generate the PID control routine; this aspect will be covered in a future application note.

Limits were placed upon the control outputs so that the signals to the processors would not exceed the physical limits of the external devices. For example, the frequency range is limited to range between 0 and 550 to correspond with the operating range of the weighbelt as we have defined it. The limits upon the liquid control valve have been set at plus and minus 10 steps since this is the maximum distance which the stepper motor can travel in any one 200 millisecond time period; increasing the possible count could result in filling the queue. This could cause the 200 millisecond time to be extended if we had to wait for the queue to empty.

Master Processor

A complete control solution to the weighbelt feeder and the liquid applicator has now been developed. The process is stand alone and resides entirely upon a single board. It can operate without requiring any access from the MULTIBUS bus, thus freeing the bus for other control, monitoring or supervisory duties.

The system developed for this application note requires a setpoint for the mass flow and a liquid ratio be provided to the control system. This information would normally be supplied by some type of bus master device. We have chosen to use the pre-configured RMX/80 BASIC-80 Interpreter to perform this task. A simple program needs to be prepared which will allow adjustment of the setpoints and monitoring of the operation of the control system.

Using BASIC will provide full disk I/O capabilities to the operator. Communicating with the

system through a CRT terminal, he can write and execute programs which use the resources of the system disk or of any of the controllers which may be present on the bus.

Two programs are required to perform this task. Since they are written in BASIC, they may easily be modified or expanded if the need should ever arise. Indeed, other programs could be written to perform other tasks, such as optimizing the control parameters.

In both programs, the parameters involved with the control operation are accessed by using the PEEK and POKE instructions. Remember that the iSBC 569 controller allows the on-board memory to be made available to other devices on the bus through the dual port mechanism. In our application, this has been done by jumpering the board such that the on-board memory beginning at location 8000H can be accessed on the bus at location 2000H. This mapping was done since the memory locations at 2000H are not used by BASIC unless requested to do so. A byte of data which is at location 827EH on the controller can be read by performing a PEEK of location 227EH. Some of the memory assignments for the controller have been shown in Figure 31.

MOD	MAINCONTROLMODULE	
829FH	SYM	MEMORY
8233H	SYM	PRLQ
825FH	SYM	CONSTANT$1
00DCH	SYM	BOUNDS2
00E6H	SYM	TIMEINTERVAL
827AH	SYM	LIQUIDFLOW
00E8H	SYM	DISTREV
8280H	SYM	MASSFLOW
8285H	SYM	LIQUIDVALVE
8288H	SYM	DUMMY
00EFH	SYM	ZERO
01ADH	SYM	PIDRUN
81F7H	SYM	IR
825DH	SYM	LIQCOUNT
8268H	SYM	CONSTANTS2
00E4H	SYM	CONTROL1
8277H	SYM	BELTSPEED
827CH	SYM	MASSSETPOINT
00E9H	SYM	CONVLENGTH
8282H	SYM	BELTCONTROL
8287H	SYM	SYSTEMRUNNING
828AH	SYM	ICW
3F00H	SYM	JUMPTABLE
8209H	SYM	PRCV
825EH	SYM	BELTCOUNT
00D4H	SYM	BOUNDS1
00E5H	SYM	CONTROL2
8278H	SYM	BELTWEIGHT
827EH	SYM	SETPOINT
00EAH	SYM	SIX
8284H	SYM	LIQUIDRATIO
00EBH	SYM	ERRORFIELD
00EDH	SYM	THOUSAND
00F1H	SYM	INITIATION

Figure 31. Selected Memory Location Assignments

The first program involves setting up the two control parameters and handling the control flag which causes the process to start and to stop. This program can be found in Figure 32.

```
10 REM THIS PROGRAM IS USED TO INPUT SETPOINTS
15 REM TO THE LIQUID CONTROL SYSTEM.
20 POKE 02287H,0
25 INPUT "ENTER MASS SETPOINT-";MS
26 IF MS > 1200 THEN 25
30 MS=CINT(MS×10/60)
35 H=INT(MS/256)
40 L=CINT(MS-H×256)
45 POKE 0227EH,L
50 POKE 0227FH,H
55 INPUT "PERCENT LIQUID-";LR
60 LR=CINT(LR)
65 IF LR > 127 THEN 55
70 POKE 02284H,LR
75 POKE 02287H,1
80 RUN "STATUS"
```

Figure 32. Basic Program for Parameter Initialization

Upon completion of the initialization program, a second program provides a display of the system operation. This program could have been an optional program which is only called when the operator desires to view the system operation. A program which provides a snapshot of the system operation is shown in Figure 33. Again, the program is interactive with the operator and can easily be modified at any time to reformat or display additional information.

IX. CONCLUSION

The purpose of this application note has been to demonstrate some of the techniques which can be used to provide a control system design solution using an intelligent slave concept. This has been done and the system has been constructed and has been found to operate as the design specified. The Intelligent Slave Concept does provide a single board solution to distributed control and certainly off-loads the master processor of control duties.

PROGRAM NAME: STATUS

```
 10 I=PEEK(0227EH)
 20 H=PEEK(0227FH)
 30 MS=((256×H)+L)×60/10
 40 L=PEEK(02278H)
 50 H=PEEK(02279H)
 60 WT=((256×H)+L)/100
 70 L=PEEK(022890H)
 80 H=PEEK(02281H)
 90 AM=((256×H)+L)×60/10
100 MT=PEEK(02294H)
110 LR=(PEEK(02284H))/100
120 LS=AM×LR
130 L=PEEK(0227AH)
140 H=PEEK(0227BH)
150 LF=((256×H)+L)/100
160 PRINT "MASS SETPOINT","WEIGHT","ACTUAL MASS","MOTION"
170 PRINT MS,WT,AM,MT
180 PRINT "LIQUID RATIO","LIQUID SET","LIQUID FLOW"
190 PRINT LR,LS,LF
200 Z=PEEK(02285H)
210 IF Z < 128 THEN 230
220 Z=256-Z
225 Z=0-Z
230 L=PEEK(02282H)
231 H=PEEK(02283H)
232 BS=((256×H)+L)×60/200
239 PRINT "STEPPER";Z, "BELT";BS
240 PRINT " "
250 PRINT " "
260 FOR N=0 to 1000
270 NEXT N
280 GO TO 10
```

Figure 33. Basic Snapshot Program

This frees the master to provide supervisory control and human interface duties.

Certainly, this concept can be expanded to encompass a broad variety of complex control situations. At the same time, there is no reason why the Intelligent Slave board could not be used to provide a single board solution to a simple control application where no interaction with other processes is required.

APPENDIX A

```
                /*******************************************
                * THIS IS THE MAIN BACKGROUND OPERATING    *
                * PROGRAM FOR THE PID CONTROL SYSTEM.      *
                *******************************************/

  1             BACKGROUND$MODULE: DO;

                /* DECLARATION OF BOARD I/O ASSIGNMENTS */
  2    1            DECLARE UPI$0$STATUS          LITERALLY 'ØE5H';
  3    1            DECLARE UPI$1$STATUS          LITERALLY 'ØE7H';
  4    1            DECLARE UPI$2$STATUS          LITERALLY 'ØE9H';

  5    1            DECLARE UPI$0$COMMAND         LITERALLY 'ØE5H';
  6    1            DECLARE UPI$1$COMMAND         LITERALLY 'ØE7H';
  7    1            DECLARE UPI$2$COMMAND         LITERALLY 'ØE9H';

  8    1            DECLARE UPI$0$DATA            LITERALLY 'ØE4H';
  9    1            DECLARE UPI$1$DATA            LITERALLY 'ØE6H';
 1Ø    1            DECLARE UPI$2$DATA            LITERALLY 'ØE8H';

 11    1            DECLARE RESET$LATCH$ADR       LITERALLY 'ØEAH';

                /* DECLARATION OF RAM TEST PARAMETERS */
 12    1            DECLARE BEGIN$RAM             LITERALLY '8ØØØH';
 13    1            DECLARE END$RAM               LITERALLY '85ØØH';
 14    1            DECLARE ZERO$PATTERN          LITERALLY 'ØØØH';
 15    1            DECLARE ONES$PATTERN          LITERALLY 'ØFFH';

                /* DECLARATION OF RESET LATCH BIT ASSIGNMENTS */
 16    1            DECLARE RESET$UPI$Ø           LITERALLY 'ØØØØØØØ1B';
 17    1            DECLARE RESET$UPI$1           LITERALLY 'ØØØØØØ1ØB';
 18    1            DECLARE RESET$UPI$2           LITERALLY 'ØØØØØ1ØØB';
 19    1            DECLARE LIGHT$LED             LITERALLY 'ØØØØ1ØØØB';
 2Ø    1            DECLARE MULTI$INTR            LITERALLY 'ØØØ1ØØØØB';

                /* DECLARATION OF ISBC 941 STATUS BITS */
 21    1            DECLARE IBF                   LITERALLY 'ØØØØØØ1ØB';
 22    1            DECLARE OBF                   LITERALLY 'ØØØØØØØ1B';

                /* DECLARATION OF ISBC 941 COMMANDS */
 23    1            DECLARE IDEN                  LITERALLY 'ØØØH';

                /* DECLARATION OF ISBC 941 IDENTIFICATION CODE */
 24    1            DECLARE SBC941                LITERALLY '41H';

                /* DECLARATION OF MEMORY TEST ADDRESS REGISTER */
 25    1            DECLARE I ADDRESS AT (87FEH);
 26    1            DECLARE MEMLOC BASED I BYTE;

                /* DECLARATION OF RESET MASKS FOR 8Ø85 PROCESSOR */
```

```
27   1              DECLARE MASKS BYTE DATA (00FH);

                 /* DECLARATION OF PL/M-80 SIM INSTRUCTION */
28   1              S$MASK: PROCEDURE (MASK) EXTERNAL;
29   2                  DECLARE MASK BYTE;
30   2                  END S$MASK;

                 /* DECLARATION OF INITIATION TASK */
31   1              INITIATION:
                        PROCEDURE EXTERNAL;
32   2                  END INITIATION;

                 /* CLEAR ISBC 941 DEVICES USING ON-BOARD RESET */
33   1              OUTPUT (RESET$LATCH$ADR) = 0;

                 /* MASK OUT THE RESET INTERRUPTS OF THE PROCESSOR */
34   1              CALL S$MASK (MASKS);

                 /* TEST MEMORY RAM LOCATIONS */
35   1              DO I = BEGIN$RAM TO END$RAM;
36   2                  MEMLOC = ZERO$PATTERN;
37   2                  DO WHILE MEMLOC <> ZERO$PATTERN;
38   3                  END;

39   2                  MEMLOC = ONES$PATTERN;
40   2                  DO WHILE MEMLOC <> ONES$PATTERN;
41   3                  END;
42   2              END;

                 /* RELEASE 941 LOCKOUT/RESET BITS */
43   1              OUTPUT (RESET$LATCH$ADR) = RESET$UPI$0 OR
                                               RESET$UPI$1 OR
                                               RESET$UPI$2 OR
                                               MULTI$INTR;

                 /* VERIFY THAT SBC941 PROCESSOR IS IN SOCKET 0 */
44   1              DO WHILE ((INPUT (UPI$0$STATUS) AND IBF) <> 0);
45   2              END;
46   1              OUTPUT (UPI$0$COMMAND) = IDEN;
47   1              DO WHILE ((INPUT (UPI$0$STATUS) AND OBF) = 0);
48   2              END;
49   1              DO WHILE (INPUT (UPI$0$DATA) <> SBC941);
50   2              END;

                 /* VERIFY THAT SBC941 PROCESSOR IS IN SOCKET 1 */
51   1              DO WHILE ((INPUT (UPI$1$STATUS) AND IBF) <> 0);
52   2              END;
53   1              OUTPUT (UPI$1$COMMAND) = IDEN;
54   1              DO WHILE ((INPUT (UPI$1$STATUS) AND OBF) = 0);
55   2              END;
56   1              DO WHILE (INPUT (UPI$1$DATA) <> SBC941);
57   2              END;
```

```
                    /* VERIFY THAT SBC941 PROCESSOR IS IN SOCKET 2 */
58    1                 DO WHILE ((INPUT (UPI$2$STATUS) AND IBF) <> 0);
59    2                 END;
60    1             OUTPUT (UPI$2$COMMAND) = IDEN;
61    1             DO WHILE ((INPUT (UPI$2$STATUS) AND OBF) = 0);
62    2             END;
63    1             DO WHILE (INPUT (UPI$2$DATA) <> SBC941);
64    2             END;

                    /* START-UP TEST OK- TURN OFF LED */
65    1             OUTPUT (RESET$LATCH$ADR) = RESET$UPI$0 OR
                                               RESET$UPI$1 OR
                                               RESET$UPI$2 OR
                                               LIGHT$LED    OR
                                               MULTI$INTR;

                    /* INITIATE THE CONTROL DEVICES */
66    1             CALL INITIATION;

                    /* PERFORM BACKGROUND TASKS */
67    1             DO WHILE 1;
68    2                 HALT;
69    2             END;

70    1         END BACKGROUND$MODULE;


MODULE INFORMATION:

    CODE AREA SIZE     = 00D4H     212D
    VARIABLE AREA SIZE = 0000H       0D
    MAXIMUM STACK SIZE = 0002H       2D
    128 LINES READ
    0 PROGRAM ERROR(S)

END OF PL/M-80 COMPILATION
```

```
          $INTVECTOR(4,3F00H)
          $PAGEWIDTH(72)
          $TITLE('MAIN CONTROL')
          /*****************************************************
          *                     MAIN$CONTROL$TASK             *
          * THIS TASK IS USED TO CONTROL THE TWO PID CONTROL  *
          * LOOPS. ONE LOOP CONTROLS THE SPEED OF A CONVEYOR  *
          * WHILE THE SECOND CONTROLS THE FLOW OF A LIQUID.   *
          * THE TASK OPERATES EACH 200 MSEC.                  *
          *                                                   *
          ******** VERSION 1.1 ******************************/

1         MAIN$CONTROL$MODULE: DO;

          /* DECLARATION OF PID RECORD SET-UP TASK */
2    1        UQPSET:
                  PROCEDURE (PR$PTR,ERROR$FLD$PTR,PRIV$PTR) EXTERNAL
     -       ;
3    2            DECLARE (PR$PTR,ERROR$FLD$PTR,PRIV$PTR) ADDRESS;
4    2            END UQPSET;

          /* DECLARATION OF PID CONTROL BITS */
5    1        UQPSCT:
                  PROCEDURE (PR$PTR,CONTROL$PTR) EXTERNAL;
6    2            DECLARE (PR$PTR,CONTROL$PTR) ADDRESS;
7    2            END UQPSCT;

          /* PROCEDURE TO SET UP PID CONSTANTS */
8    1        UQPSCN:
                  PROCEDURE (PR$PTR,CONSTANT$PTR) EXTERNAL;
9    2            DECLARE (PR$PTR,CONSTANT$PTR) ADDRESS;
10   2            END UQPSCN;

          /* DEFINE THE DEFAULT ERROR HANDLER */
11   1        UQPSBD:
                  PROCEDURE (PR$PTR,BOUND$PTR) EXTERNAL;
12   2            DECLARE (PR$PTR,BOUND$PTR) ADDRESS;
13   2            END UQPSBD;

          /* PROCEDURE TO CHANGE THE TIME INTERVAL */
14   1        UQPSTI:
                  PROCEDURE (PR$PTR,TIME$INTERVAL$PTR) EXTERNAL;
15   2            DECLARE (PR$PTR,TIME$INTERVAL$PTR) ADDRESS;
16   2            END UQPSTI;

          /* DECLARATION OF THE PID CONTROL PROGRAM */
17   1        UQPPID:
                  PROCEDURE (PR$PTR,IR$PTR) EXTERNAL;
18   2            DECLARE (PR$PTR,IR$PTR) ADDRESS;
19   2            END UQPPID;
```

```
                /* DECLARATION OF WEIGHBELT SPEED INTERFACE */
20    1             WEIGHBELT$SPEED:
                        PROCEDURE BYTE EXTERNAL;
21    2                 END WEIGHBELT$SPEED;

                /* DECLARATION OF WEIGHBELT WEIGHT INTERFACE */
22    1             WEIGHBELT$WEIGHT:
                        PROCEDURE ADDRESS EXTERNAL;
23    2                 END WEIGHBELT$WEIGHT;

                /* DECLARATION OF LIQUID FLOW RATE INTERFACE */
24    1             LIQUID$FLOW$RATE:
                        PROCEDURE ADDRESS EXTERNAL;
25    2                 END LIQUID$FLOW$RATE;

                /* DECLARATION OF WEIGHBELT MOTOR DRIVE INTERFACE */
26    1             WEIGHBELT$MOTOR$DRIVE:
                        PROCEDURE (SPEED) EXTERNAL;
27    2                 DECLARE SPEED ADDRESS;
28    2                 END WEIGHBELT$MOTOR$DRIVE;

                /* DECLARATION OF LIQUID VALVE INTERFACE */
29    1             LIQUID$VALVE$POSITION:
                        PROCEDURE (POSITION) EXTERNAL;
30    2                 DECLARE POSITION BYTE;
31    2                 END LIQUID$VALVE$POSITION;

                /* DECLARATION OF PROCESSOR Ø INITIALIZATION MODULE */
32    1             PROCESSOR$Ø$INITIALIZATION:
                        PROCEDURE EXTERNAL;
33    2                 END PROCESSOR$Ø$INITIALIZATION;

                /* DECLARATION OF PROCESSOR 1 INITIALIZATION MODULE */
34    1             PROCESSOR$1$INITIALIZATION:
                        PROCEDURE EXTERNAL;
35    2                 END PROCESSOR$1$INITIALIZATION;

                /* DECLARATION OF PROCESSOR 2 INITIALIZATION MODULE */
36    1             PROCESSOR$2$INITIALIZATION:
                        PROCEDURE EXTERNAL;
37    2                 END PROCESSOR$2$INITIALIZATION;

                /* DECLARATION OF PIT COUNTER 1 INITIALIZATION */
38    1             COUNTER$1$INITIALIZATION:
                        PROCEDURE EXTERNAL;
39    2                 END COUNTER$1$INITIALIZATION;

                /* DECLARATION OF PIT COUNTER 2 INITIALIZATION */
40    1             COUNTER$2$INITIALIZATION:
                        PROCEDURE EXTERNAL;
41    2                 END COUNTER$2$INITIALIZATION;
```

```
                    /* DECLARATION OF FSP UNSIGNED LOAD PROCEDURES */
42    1             MQULD1: PROCEDURE (IR$PTR,VALUE$PTR) EXTERNAL;
43    2                 DECLARE (IR$PTR,VALUE$PTR) ADDRESS;
44    2                 END MQULD1;
45    1             MQULD2: PROCEDURE (IR$PTR,VALUE$PTR) EXTERNAL;

46    2                 DECLARE (IR$PTR,VALUE$PTR) ADDRESS;
47    2                 END MQULD2;


                    /* DECLARATION OF FSP UNSIGNED MULTIPLY PROCEDURE */
48    1             MQUML1: PROCEDURE (IR$PTR,VALUE$PTR) EXTERNAL;
49    2                 DECLARE (IR$PTR,VALUE$PTR) ADDRESS;
50    2                 END MQUML1;
51    1             MQUML2: PROCEDURE (IR$PTR,VALUE$PTR) EXTERNAL;
52    2                 DECLARE (IR$PTR,VALUE$PTR) ADDRESS;
53    2                 END MQUML2;


                    /* DECLARATION OF FSP UNSIGNED DIVIDE PROCEDURE */
54    1             MQUDV1: PROCEDURE (IR$PTR,VALUE$PTR) EXTERNAL;
55    2                 DECLARE (IR$PTR,VALUE$PTR) ADDRESS;
56    2                 END MQUDV1;
57    1             MQUDV2: PROCEDURE (IR$PTR,VALUE$PTR) EXTERNAL;
58    2                 DECLARE (IR$PTR,VALUE$PTR) ADDRESS;
59    2                 END MQUDV2;


                    /* DECLARATION OF FSP SIGNED DIVIDE PROCEDURE */
60    1             MQSDV1: PROCEDURE (IR$PTR,VALUE$PTR) EXTERNAL;
61    2                 DECLARE (IR$PTR,VALUE$PTR) ADDRESS;
62    2                 END MQSDV1;
63    1             MQSDV2: PROCEDURE (IR$PTR,VALUE$PTR) EXTERNAL;
64    2                 DECLARE (IR$PTR,VALUE$PTR) ADDRESS;
65    2                 END MQSDV2;


                    /* DECLARTATION OF FSP SIGNED STORE PROCEDURE */
66    1             MQSST2: PROCEDURE (IR$PTR,VALUE$PTR) EXTERNAL;
67    2                 DECLARE (IR$PTR,VALUE$PTR) ADDRESS;
68    2                 END MQSST2;


                    /* DECLARATION OF FSP SIGNED LOAD PROCEDURE */
69    1             MQSLD2: PROCEDURE (IR$PTR,VALUE$PTR) EXTERNAL;
70    2                 DECLARE (IR$PTR,VALUE$PTR) ADDRESS;
71    2                 END MQSLD2;


                    /* DECLARATION OF FSP SIGNED SUBTRACT PROCEDURE */
72    1             MQSSB2: PROCEDURE (IR$PTR,VALUE$PTR) EXTERNAL;
73    2                 DECLARE (IR$PTR,VALUE$PTR) ADDRESS;
74    2                 END MQSSB2;


                    /* DECLARATION OF FSP UNSIGNED STORE PROCEDURE */
75    1             MQUST1: PROCEDURE (IR$PTR,VALUE$PTR) EXTERNAL;
76    2                 DECLARE (IR$PTR,VALUE$PTR) ADDRESS;
77    2                 END MQUST1;
78    1             MQUST2: PROCEDURE (IR$PTR,VALUE$PTR) EXTERNAL;
79    2                 DECLARE (IR$PTR,VALUE$PTR) ADDRESS;
80    2                 END MQUST2;
```

```
                /* DECLARATION OF FSP SIGNED MULTIPLY PROCEDURE */
81    1             MQSML1: PROCEDURE (IR$PTR,VALUE$PTR) EXTERNAL;
82    2                 DECLARE (IR$PTR,VALUE$PTR) ADDRESS;
83    2                 END MQSML1;
          $EJECT
          /******************************************
          *   DATA STORAGE AREAS FOR THE PID CONTROL *
          ******************************************/

                /* DEFINITION OF LIMITATION CONSTANTS */
84    1             DECLARE MAX$MOTOR$SPEED        LITERALLY '550';
85    1             DECLARE MIN$MOTOR$SPEED        LITERALLY '0';
86    1             DECLARE MAX$VALVE$MOVEMENT     LITERALLY '10';
87    1             DECLARE MIN$VALVE$MOVEMENT     LITERALLY '-10';

                /* DEFINITION OF PID PARAMETER COEFFICIENTS */
88    1             DECLARE FEEDER$C0              LITERALLY '1';
89    1             DECLARE FEEDER$C1              LITERALLY '1';
90    1             DECLARE FEEDER$C2              LITERALLY '1';
91    1             DECLARE FEEDER$C3              LITERALLY '1';
92    1             DECLARE FEEDER$TIME$INTERVAL   LITERALLY '1';
93    1             DECLARE FEEDER$SCALE$FACTOR    LITERALLY '1';

94    1             DECLARE LIQUID$C0             LITERALLY '1';
95    1             DECLARE LIQUID$C1             LITERALLY '1';
96    1             DECLARE LIQUID$C2             LITERALLY '1';
97    1             DECLARE LIQUID$C3             LITERALLY '1';
98    1             DECLARE LIQUID$TIME$INTERVAL  LITERALLY '1';
99    1             DECLARE LIQUID$SCALE$FACTOR   LITERALLY '10';

                /* DEFINITION OF RESET LATCH PARAMETERS */
100   1             DECLARE RESET$LATCH$ADR        LITERALLY '0EAH';
101   1             DECLARE INDICATOR$ON           LITERALLY '07H';
102   1             DECLARE INDICATOR$OFF          LITERALLY '0FH';

                /* RESERVE 18 BYTES FOR THE INTEGER RECORD */
103   1             DECLARE IR (18) BYTE PUBLIC;

                /* RESERVE 42 BYTES FOR EACH PID RECORD */
104   1             DECLARE PRCV (42) BYTE;
105   1             DECLARE PRLQ (42) BYTE;

                /* RESERVE SPACE FOR COUNTER DATA */
106   1             DECLARE (LIQ$COUNT,BELT$COUNT) BYTE PUBLIC;

                /* RESERVE 12 BYTES FOR EACH CONSTANT ARRAY */
107   1             DECLARE CONSTANTS1 STRUCTURE (
                        C0   ADDRESS,
                        C1   ADDRESS,
                        C2   ADDRESS,
                        C3   ADDRESS,
                        DT   ADDRESS,
                        S    ADDRESS );
```

```
108   1         DECLARE CONSTANTS2 STRUCTURE (
                    C0   ADDRESS,
                    C1   ADDRESS,
                    C2   ADDRESS,
                    C3   ADDRESS,
                    DT   ADDRESS,
                    S    ADDRESS );
                /* RESERVE 8 BYTES FOR EACH BOUNDS ARRAY */
109   1         DECLARE BOUNDS1 (4) ADDRESS DATA (
                    000H,
                    000H,
                    MAX$MOTOR$SPEED,
                    MIN$MOTOR$SPEED );
110   1         DECLARE BOUNDS2 (4) ADDRESS DATA (
                    000D,
                    000D,
                    MAX$VALVE$MOVEMENT,
                    MIN$VALVE$MOVEMENT );

                /* RESERVE 1 BYTE FOR EACH CONTROL BYTE */
111   1         DECLARE CONTROL1 BYTE DATA (073H);
112   1         DECLARE CONTROL2 BYTE DATA (053H);

                /* DECLARE TIME INTERVAL */
113   1         DECLARE TIME$INTERVAL ADDRESS DATA (1);

                /* RESERVE SPACE FOR THE CURRENT BELT SPEED */
114   1         DECLARE BELT$SPEED BYTE;

                /* RESERVE SPACE FOR THE CURRENT BELT WEIGHT */
115   1         DECLARE BELT$WEIGHT ADDRESS;

                /* RESERVE SPACE FOR THE LIQUID FLOW */
116   1         DECLARE LIQUID$FLOW ADDRESS;

                /* RESERVE SPACE FOR THE EFFECTIVE SETPOINT */
117   1         DECLARE MASS$SETPOINT ADDRESS;

                /* RESERVE SPACE FOR THE DESIRED SETPOINT */
118   1         DECLARE SET$POINT ADDRESS;

                /* RESERVE SPACE FOR THE DISTANCE OF BELT PER REVOLUTION
      -         */
119   1         DECLARE DIST$REV BYTE DATA (100);

                /* DEFINE THE CONVEYOR LENGTH */
120   1         DECLARE CONV$LENGTH BYTE DATA (200);

                /* DEFINE THE CONSTANT SIX */
121   1         DECLARE SIX BYTE DATA (6);

                /* RESERVE STORAGE FOR ACTUAL CURRENT MASS FLOW */
122   1         DECLARE MASS$FLOW ADDRESS;
```

```
            /* RESERVE SPACE FOR BELT CONTROL OUTPUT */
123    1        DECLARE BELT$CONTROL ADDRESS;

            /* RESERVE SPACE FOR LIQUID RATIO */
124    1        DECLARE LIQUID$RATIO BYTE;

            /* RESERVE SPACE FOR LIQUID CONTROL OUTPUT */
125    1        DECLARE LIQUID$VALVE ADDRESS;

            /* RESERVE SPACE FOR RUN/HALT CONTROL */
126    1        DECLARE SYSTEM$RUNNING BYTE PUBLIC;

            /* RESERVE SPACE FOR ERROR FIELD */
127    1        DECLARE ERROR$FIELD ADDRESS DATA (0F800H);
128    1        DECLARE DUMMY ADDRESS;

          /* RESERVE SPACE FOR PIC ICW BYTE */
129    1        DECLARE ICW BYTE;

            /* DEFINE CONSTANT 1000 */
130    1        DECLARE THOUSAND ADDRESS DATA (1000);

            /* DEFINE CONSTANT 0 */
131    1        DECLARE ZERO ADDRESS DATA (0);

            /* DEFINE INTERRUPT JUMP TABLE */
132    1        DECLARE JUMP$TABLE BYTE AT (3F00H);

          /* DECLARATION OF PIC ADDRESSES ON ISBC 569 BOARD */
133    1        DECLARE PIC$ICW1$PTR     LITERALLY '0ECH';
134    1        DECLARE PIC$ICW2$PTR     LITERALLY '0EDH';
135    1        DECLARE PIC$INT$MASK$PTR LITERALLY '0EDH';

          /* DECLARATION OF PIC CONSTANTS */
136    1        DECLARE CLR$LOW$BITS     LITERALLY '0E0H';
137    1        DECLARE INTERVAL$4       LITERALLY '016H';
138    1        DECLARE INTERRUPT$MASK   LITERALLY '0F4H';
            $EJECT
            /*****************************************
            * INITIALIZE PROGRAM AT START-UP OF SYSTEM *
            * THIS PROCEDURE IS CALLED AT START-UP       *
            *****************************************/

139    1    INITIATION: PROCEDURE PUBLIC;

            /* DISABLE THE INTERRUPTS */
140    2        DISABLE;

            /* INITIALIZE PID RECORD */
141    2        CALL UQPSET (.PRCV,.ERROR$FIELD,.DUMMY);
142    2        CALL UQPSET (.PRLQ,.ERROR$FIELD,.DUMMY);
```

```
                        /* INITIALIZE THE CONTROL BITS */
143    2                CALL UQPSCT (.PRCV,.CONTROL1);
144    2                CALL UQPSCT (.PRLQ,.CONTROL2);

                        /* SET UP THE PID CONSTANTS */
145    2                CONSTANTS1.CØ = FEEDER$CØ;
146    2                CONSTANTS1.C1 = FEEDER$C1;
147    2                CONSTANTS1.C2 = FEEDER$C2;
148    2                CONSTANTS1.C3 = FEEDER$C3;
149    2                CONSTANTS1.DT = FEEDER$TIME$INTERVAL;
15Ø    2                CONSTANTS1.S  = FEEDER$SCALE$FACTOR;

151    2                CONSTANTS2.CØ = LIQUID$CØ;
152    2                CONSTANTS2.C1 = LIQUID$C1;
153    2                CONSTANTS2.C2 = LIQUID$C2;
154    2                CONSTANTS2.C3 = LIQUID$C3;
155    2                CONSTANTS2.DT = LIQUID$TIME$INTERVAL;
156    2                CONSTANTS2.S  = LIQUID$SCALE$FACTOR;

                        /* CLEAR SETPOINTS */
157    2                SETPOINT = Ø;
158    2                LIQUID$RATIO = Ø;
159    2                SYSTEM$RUNNING = Ø;

                        /* INITIALIZE THE CONSTANTS */
16Ø    2                CALL UQPSCN (.PRCV,.CONSTANTS1);
161    2                CALL UQPSCN (.PRLQ,.CONSTANTS2);

                        /* INITIALIZE THE BOUNDS */
162    2                CALL UQPSBD (.PRCV,.BOUNDS1);
163    2                CALL UQPSBD (.PRLQ,.BOUNDS2);

                        /* SET THE TIME INTERVAL */
164    2                CALL UQPSTI (.PRCV,.TIME$INTERVAL);
165    2                CALL UQPSTI (.PRLQ,.TIME$INTERVAL);

                        /* INITIALIZE PROCESSOR Ø */
166    2                CALL PROCESSOR$Ø$INITIALIZATION;

                        /* INITIALIZE PROCESSOR 1 */
167    2                CALL PROCESSOR$1$INITIALIZATION;
                        /* INITIALIZE PROCESSOR 2 */
168    2                CALL PROCESSOR$2$INITIALIZATION;

                        /* INITIALIZE  COUNTER 1 */
169    2                CALL COUNTER$1$INITIALIZATION;

                        /* INITIALIZE  COUNTER 2 */
17Ø    2                CALL COUNTER$2$INITIALIZATION;

                        /* INITIALIZE INTERRUPT CONTROLLER */
171    2                ICW = (LOW (.JUMP$TABLE) AND
                            CLR$LOW$BITS ) OR
                            INTERVAL$4 ;
172    2                OUTPUT (PIC$ICW1$PTR) = ICW;
```

```
173   2            ICW = HIGH (.JUMP$TABLE);
174   2            OUTPUT (PIC$ICW2$PTR) = ICW;

             /* SET INTERRUPT MASKS */
175   2            OUTPUT (PIC$INT$MASK$PTR) = INTERRUPT$MASK;

             /* ENABLE INTERRUPTS */
176   2            ENABLE;

             /* RETURN TO MAIN PROGRAM */
177   2            RETURN;

178   2      END INITIATION;
             $EJECT
             /******************************************************
             *   THIS IS THE PID CONTROL ROUTINE.  IT IS ENTERED *
             *   EACH 200 MILLISECONDS THROUGH AN INTERRUPT GEN- *
             *   ERATED BY THE FREQUENCY COUNTER UPI AND SENT TO  *
             *   INTERRUPT 3.                                     *
             ******************************************************/


179   1      PIDRUN:  PROCEDURE INTERRUPT 3 PUBLIC;

             /* TURN THE LED INDICATOR ON */
180   2            OUTPUT(RESET$LATCH$ADR) = INDICATOR$ON;

             /* GET WEIGHBELT WEIGHT */
181   2            BELT$WEIGHT=WEIGHBELT$WEIGHT;

             /* GET LIQUID FLOW RATE */
182   2            LIQUID$FLOW=LIQUID$FLOW$RATE;

             /* CONTROL START-STOP RAMP */
183   2            IF SYSTEM$RUNNING
                     THEN MASS$SETPOINT=SETPOINT;
185   2              ELSE MASS$SETPOINT=0;

             /* DETERMINE ACTUAL MASS FLOW ON WEIGHBELT */
186   2            CALL MQULD2(.IR,.BELT$CONTROL);
187   2            CALL MQUML2(.IR,.BELT$WEIGHT);
188   2            CALL MQUML1(.IR,.DIST$REV);
189   2            CALL MQUDV1(.IR,.CONV$LENGTH);
190   2            CALL MQSDV2(.IR,.THOUSAND);
191   2            CALL MQSST2(.IR,.MASS$FLOW);

             /* COMPUTE ERROR SIGNAL ON WEIGHBELT */
192   2            CALL MQSLD2(.IR,.MASS$SETPOINT);
193   2            CALL MQSSB2(.IR,.MASS$FLOW);

             /* HANDLE PID BELT CONTROL ALGORITHM */
194   2            CALL UQPPID(.PRCV,.IR);

             /* STORE OUTPUT SIGNAL */
195   2            CALL MQUST2(.IR,.BELT$CONTROL);
```

```
                    /* COMPUTE LIQUID SETPOINT */
196   2                 CALL MQSLD2(.IR,.MASS$FLOW);
197   2                 CALL MQSML1(.IR,.LIQUID$RATIO);
198   2                 CALL MQSML1(.IR,.SIX);


                    /* VERIFY THAT WEIGHBELT IS MOVING */
199   2                 IF WEIGHBELT$SPEED = 0
                        THEN CALL MQULD2(.IR,.ZERO);


                    /* COMPUTE LIQUID ERROR */
201   2                 CALL MQSSB2(.IR,.LIQUID$FLOW);


                    /* HANDLE PID LIQUID CONTROL */
202   2                 CALL UQPPID(.PRLQ,.IR);

                    /* STORE OUTPUT SIGNAL */
203   2                 CALL MQUST1(.IR,.LIQUID$VALVE);


                    /* OUTPUT WEIGHBELT CONTROL SIGNAL */
204   2                 CALL WEIGHBELT$MOTOR$DRIVE (BELT$CONTROL);


                    /* OUTPUT FLOW CONTROL SIGNAL */
205   2                 CALL LIQUID$VALVE$POSITION (LIQUID$VALVE);


                    /* SEND END OF INTERRUPT TO 8259 CONTROLLER */
206   2                 OUTPUT(0ECH)=020H;


                    /* TURN THE LED INDICATOR OFF */
207   2                 OUTPUT (RESET$LATCH$ADR) = INDICATOR$OFF;


                    /* RETURN FROM CONTROL TASK */
208   2                 RETURN;
209   2                 END PIDRUN;
210   1         END;



MODULE INFORMATION:

     CODE AREA SIZE      = 01C1H     449D
     VARIABLE AREA SIZE = 0094H     148D
     MAXIMUM STACK SIZE = 000AH      10D
     465 LINES READ
     0 PROGRAM ERROR(S)

END OF PL/M-80 COMPILATION
```

ISIS-II PL/M-80 V3.1 COMPILATION OF MODULE PROCESSORINITIALIZATIONMODULE
OBJECT MODULE PLACED IN :F1:SBC941.OBJ
COMPILER INVOKED BY: PLM80 :F1:SBC941.PLM DEBUG PAGEWIDTH(72) TITLE('PR
               -OCESSOR INITIALIZATION')


```
                /***********************************************
                * THIS PROGRAM IS USED TO INITIALIZE THE ISBC *
                * 941 PROCESSOR INSTALLED IN SOCKET 0.   THE  *
                * DEVICE WILL OPERATE IN THE FREQUENCY OUTPUT  *
                * MODE.                                        *
                ***********************************************/

 1              PROCESSOR$INITIALIZATION$MODULE: DO;

                /* DECLARATION OF ADDRESSES */
 2      1           DECLARE UPI$0$STATUS    LITERALLY '0E5H';
 3      1           DECLARE UPI$0$COMMAND   LITERALLY '0E5H';
 4      1           DECLARE UPI$0$DATA      LITERALLY '0E4H';

 5      1           DECLARE UPI$1$STATUS    LITERALLY '0E7H';
 6      1           DECLARE UPI$1$COMMAND   LITERALLY '0E7H';
 7      1           DECLARE UPI$1$DATA      LITERALLY '0E6H';

 8      1           DECLARE UPI$2$STATUS    LITERALLY '0E9H';
 9      1           DECLARE UPI$2$COMMAND   LITERALLY '0E9H';
10      1           DECLARE UPI$2$DATA      LITERALLY '0E8H';

                /* DECLARATION OF ISBC 941 COMMANDS */
11      1           DECLARE SETP1           LITERALLY '00BH';
12      1           DECLARE CLRP1           LITERALLY '00DH';
13      1           DECLARE CLRP2           LITERALLY '00EH';
14      1           DECLARE PAUSE           LITERALLY '005H';
15      1           DECLARE LOOP            LITERALLY '004H';
16      1           DECLARE INITPF          LITERALLY '002H';
17      1           DECLARE PACIFY          LITERALLY '001H';
18      1           DECLARE ENFLAG          LITERALLY '006H';

                /* DECLARATION OF ISBC 941 STATUS BITS */
19      1           DECLARE RFC             LITERALLY '080H';
20      1           DECLARE IBF             LITERALLY '002H';
21      1           DECLARE QF              LITERALLY '010H';

                /* DECLARATION OF ISBC 941 #0 INITIALIZATION DATA */
22      1           DECLARE FREQ            LITERALLY '0B5H';
23      1           DECLARE SF              LITERALLY '000H';
24      1           DECLARE OUTPUT$ENABLE0  LITERALLY '001H';
25      1           DECLARE INITIAL$STATE   LITERALLY '000H';
26      1           DECLARE DELAY           LITERALLY '001H';
27      1           DECLARE PERIOD          LITERALLY '000H';
28      1           DECLARE INITIAL$OUTPUT  LITERALLY '00EH';
```

```
                    /* DECLARATION OF INTERVAL TIMER PARAMETERS */
29    1                 DECLARE PIT$0$MODE     LITERALLY '016H';
30    1                 DECLARE PIT$0$INTERVAL LITERALLY '00EH';
31    1                 DECLARE PIT$0$MODE$WRD LITERALLY '0E3H';
32    1                 DECLARE PIT$0$COUNT    LITERALLY '0E0H';

                    /* DECLARATION OF COUNTER LOCATIONS */
33    1                 DECLARE (LIQ$COUNT,BELT$COUNT) BYTE EXTERNAL;

                    /* DECLARATION OF ISBC 941 PRIMARY DATA */
34    1                 DECLARE INIT$0$TABLE(6) BYTE DATA (
                            FREQ,
                            SF,
                            OUTPUT$ENABLE0,
                            INITIAL$STATE,
                            DELAY,
                            PERIOD   );

                    /* DECLARATION OF MISC PARAMETERS */
35    1                 DECLARE I BYTE;


                    /***********************************************
                    *          INITIALIZATION PROGRAM BODY        *
                    ***********************************************/

36    1             PROCESSOR$0$INITIALIZATION: PROCEDURE PUBLIC;

                    /* INITIALIZE COUNTER 0 FOR 10 MICROSECONDS */
37    2                 OUTPUT(PIT$0$MODE$WRD)=PIT$0$MODE;
38    2                 OUTPUT(PIT$0$COUNT)=PIT$0$INTERVAL;

                    /* VERIFY THAT PROCESSOR IS RESET */
39    2                 DO WHILE ((INPUT(UPI$0$STATUS) AND RFC) = 0);
40    3                     DO WHILE ((INPUT(UPI$0$STATUS) AND IBF) <> 0);
41    4                     END;
42    3                     OUTPUT(UPI$0$COMMAND)=PACIFY;
43    3                 END;

                    /* REQUEST PRIMARY FUNCTION */
44    2                 DO WHILE ((INPUT(UPI$0$STATUS) AND IBF) <> 0);
45    3                 END;
46    2                 OUTPUT(UPI$0$COMMAND)= INITPF;

                    /* LOAD INITIAL PARAMETERS */
47    2                 DO I=0 TO 5;
48    3                     DO WHILE ((INPUT(UPI$0$STATUS) AND IBF) <> 0);
49    4                     END;
50    3                     OUTPUT(UPI$0$DATA)=INIT$0$TABLE(I);
51    3                 END;

                    /* TERMINATE PARAMETER LOADING */
52    2                 DO WHILE ((INPUT(UPI$0$STATUS) AND IBF) <> 0);
53    3                 END;
54    2                 OUTPUT(UPI$0$COMMAND)=PAUSE;
```

```
                    /* START FREQUENCY FUNCTION */
55    2                 DO WHILE ((INPUT(UPI$0$STATUS) AND IBF) <>0);
56    3                 END;
57    2                 OUTPUT(UPI$0$COMMAND)=LOOP;

                    /* SET UNUSED BITS TO ALLOW EXPANSION */

58    2                 DO WHILE ((INPUT(UPI$0$STATUS) AND IBF) <> 0);
59    3                 END;
60    2                 OUTPUT(UPI$0$COMMAND)=CLRP2;

61    2                 DO WHILE ((INPUT(UPI$0$STATUS) AND IBF) <> 0);
62    3                 END;
63    2                 OUTPUT(UPI$0$DATA)=INITIAL$OUTPUT;

                    /* RETURN TO CALLING PROGRAM */
64    2                 RETURN;

65    2             END PROCESSOR$0$INITIALIZATION;
                   $EJECT
                   /***********************************************
                   * THIS PROCEDURE IS USED TO INITIALIZE THE ISBC *
                   * 941 PROCESSOR INSTALLED IN SOCKET 1.   THE DE- *
                   * VICE WILL OPERATE IN THE  FCOUNT,  HIGH  FRE- *
                   * QUENCY INPUT MODE.                            *
                   ***********************************************/


                   /* DEFINE INITIALIZATION PARAMETERS */
66    1                 DECLARE FCOUNT                LITERALLY '033H';
67    1                 DECLARE INPUT$SELECT          LITERALLY '001H';
68    1                 DECLARE OUTPUT$ENABLE$1       LITERALLY '001H';
69    1                 DECLARE SAMPLING$INTERVAL     LITERALLY '009H';
70    1                 DECLARE INITIAL$STATE$1       LITERALLY '0E1H';

                   /* DECLARE PARAMETER INITIALIZATION TABLE */
71    1                 DECLARE INIT$1$TABLE(4) BYTE DATA (
                            FCOUNT,
                            INPUT$SELECT,
                            OUTPUT$ENABLE$1,
                            SAMPLING$INTERVAL    );


                   /***********************************************
                   *                 INITIALIZATION BODY          *
                   ***********************************************/

72    1            PROCESSOR$1$INITIALIZATION: PROCEDURE PUBLIC;

                   /* VERIFY THAT PROCESSOR IS RESET */
73    2                 DO WHILE ((INPUT(UPI$1$STATUS) AND RFC) = 0);
74    3                     DO WHILE ((INPUT(UPI$1$STATUS) AND IBF) <> 0);
75    4                     END;
76    3                     OUTPUT(UPI$1$COMMAND)=PACIFY;
77    3                 END;
```

```
                    /* REQUEST PRIMARY FUNCTION */
78   2                  DO WHILE ((INPUT(UPI$1$STATUS) AND IBF) <> 0);
79   3                  END;
80   2                  OUTPUT(UPI$1$COMMAND)=INITPF;

                    /* LOAD INITIAL PARAMETERS */
81   2                  DO I=0 TO 3;
82   3                      DO WHILE ((INPUT(UPI$1$STATUS) AND IBF) <> 0);
83   4                      END;
84   3                      OUTPUT(UPI$1$DATA)=INIT$1$TABLE(I);
85   3                  END;

                    /* TERMINATE PARAMETER LOADING */
86   2                  DO WHILE ((INPUT(UPI$1$STATUS) AND IBF) <> 0);
87   3                  END;
88   2                  OUTPUT(UPI$1$COMMAND)=PAUSE;

                    /* SET UNUSED BITS HIGH FOR SPARE ENABLES */
89   2                  DO WHILE ((INPUT(UPI$1$STATUS) AND IBF) <> 0);
90   3                  END;

91   2                  OUTPUT(UPI$1$COMMAND)=SETP1;
92   2                  DO WHILE ((INPUT(UPI$1$STATUS) AND IBF) <> 0);
93   3                  END;
94   2                  OUTPUT(UPI$1$DATA)=INITIAL$STATE$1;

                    /* START FREQUENCY COUNT OPERATION */
95   2                  DO WHILE ((INPUT(UPI$1$STATUS) AND IBF) <> 0);
96   3                  END;
97   2                  OUTPUT(UPI$1$COMMAND)=LOOP;

                    /* RETURN TO CALLING PROGRAM */
98   2                  RETURN;

99   2              END PROCESSOR$1$INITIALIZATION;

                $EJECT
                /*************************************************
                * THIS PROCEDURE IS USED TO INITIALIZE THE ISBC *
                * 941 INSTALLED IN SOCKET 2. THE DEVICE WILL BE *
                * OPERATED AS A STEPPER MOTOR DRIVER.           *
                *************************************************/

                    /* DEFINE INITIALIZATION PARAMETERS */
100  1              DECLARE STEPPER              LITERALLY '017H';
101  1              DECLARE SCALE$FACTOR         LITERALLY '0DFH';
102  1              DECLARE OUTPUT$ENABLE$2      LITERALLY '0F0H';
103  1              DECLARE OUTPUT$POLARITY      LITERALLY '050H';
104  1              DECLARE COMMON$PERIOD        LITERALLY '004H';
105  1              DECLARE P20$TRAN1            LITERALLY '000H';
106  1              DECLARE P20$TRAN2            LITERALLY '000H';
107  1              DECLARE P21$TRAN1            LITERALLY '000H';
108  1              DECLARE P21$TRAN2            LITERALLY '000H';
109  1              DECLARE P22$TRAN1            LITERALLY '000H';
110  1              DECLARE P22$TRAN2            LITERALLY '000H';
```

```
111   1          DECLARE  P23$TRAN1              LITERALLY '000H';
112   1          DECLARE  P23$TRAN2              LITERALLY '000H';
113   1          DECLARE  P24$TRAN1              LITERALLY '000H';
114   1          DECLARE  P24$TRAN2              LITERALLY '002H';
115   1          DECLARE  P25$TRAN1              LITERALLY '000H';
116   1          DECLARE  P25$TRAN2              LITERALLY '002H';
117   1          DECLARE  P26$TRAN1              LITERALLY '001H';
118   1          DECLARE  P26$TRAN2              LITERALLY '003H';
119   1          DECLARE  P27$TRAN1              LITERALLY '001H';
120   1          DECLARE  P27$TRAN2              LITERALLY '003H';

121   1          DECLARE  CLR$LOW$PORT           LITERALLY '00FH';


      /* DECLARE PARAMETER INITIALIZATION TABLE */
122   1          DECLARE INIT$2$TABLE(21) BYTE DATA (
                    STEPPER,
                    SCALE$FACTOR,
                    OUTPUT$ENABLE$2,
                    OUTPUT$POLARITY,
                    COMMON$PERIOD,
                    P20$TRAN1,
                    P20$TRAN2,
                    P21$TRAN1,
                    P21$TRAN2,
                    P22$TRAN1,
                    P22$TRAN2,
                    P23$TRAN1,
                    P23$TRAN2,
                    P24$TRAN1,
                    P24$TRAN2,
                    P25$TRAN1,
                    P25$TRAN2,
                    P26$TRAN1,
                    P26$TRAN2,
                    P27$TRAN1,
                    P27$TRAN2    );
      /**************************************************
      *                 INITIALIZATION BODY             *
      **************************************************/

123   1    PROCESSOR$2$INITIALIZATION: PROCEDURE PUBLIC;

           /* VERIFY THAT PROCESSOR IS RESET */
124   2        DO WHILE ((INPUT(UPI$2$STATUS) AND RFC) = 0);
125   3            DO WHILE ((INPUT(UPI$2$STATUS) AND IBF) <> 0);
126   4            END;
127   3            OUTPUT(UPI$2$COMMAND)=PACIFY;
128   3        END;

           /* REQUEST PRIMARY FUNCTION */
129   2        DO WHILE ((INPUT(UPI$2$STATUS) AND IBF) <> 0);
130   3        END;
131   2        OUTPUT(UPI$2$COMMAND)=INITPF;
```

```
                /* LOAD INITIAL PARAMETERS */
132    2            DO I=Ø TO 20;
133    3                DO WHILE ((INPUT(UPI$2$STATUS) AND IBF) <> Ø);
134    4                END;
135    3                OUTPUT(UPI$2$DATA)=INIT$2$TABLE(I);
136    3            END;

                /* TERMINATE PARAMETER LOADING */
137    2            DO WHILE ((INPUT(UPI$2$STATUS) AND IBF) <> Ø);
138    3            END;
139    2            OUTPUT(UPI$2$COMMAND)=PAUSE;

                /* START STEPPER CONTROLLER OPERATION */
140    2            DO WHILE ((INPUT(UPI$2$STATUS) AND IBF) <> Ø);
141    3            END;
142    2            OUTPUT(UPI$2$COMMAND)=LOOP;

                /* SET UNUSED BITS LOW TO ENABLE GENERAL FUNCTIONS */
143    2            DO WHILE ((INPUT(UPI$2$STATUS) AND IBF) <> Ø);
144    3            END;
145    2            OUTPUT(UPI$2$COMMAND)=CLRP1;
146    2            DO WHILE ((INPUT(UPI$2$STATUS) AND IBF) <> Ø);
147    3            END;
148    2            OUTPUT(UPI$2$DATA)=CLR$LOW$PORT;

                /* RETURN TO CALLING PROGRAM */
149    2            RETURN;

150    2        END PROCESSOR$2$INITIALIZATION;
                $EJECT
                /************************************************
                * THIS PROCEDURE IS USED TO INITIALIZE COUNTER *
                * 1 TO PERFORM AS AN EIGHT BIT BINARY COUNTER  *
                * WHICH WILL BE USED TO MEASURE THE BELT SPEED.*
                ************************************************/

151    1        COUNTER$1$INITIALIZATION: PROCEDURE PUBLIC;

                /* INITIALIZE COUNTER 1 FOR EIGHT BIT COUNTING */
152    2            LIQ$COUNT = Ø;

                /* RETURN TO CALLING PROGRAM */
153    2            RETURN;

154    2        END COUNTER$1$INITIALIZATION;
                $EJECT
                /************************************************
                * THIS PROCEDURE IS USED TO INITIALIZE COUNTER *
                * 2 TO PERFORM AS AN EIGHT BIT  BINARY COUNTER *
                * WHICH  WILL BE  USED TO  MEASURE  THE LIQUID *
                * FLOW THROUGH THE METER.                      *
                ************************************************/
```

```
155    1         COUNTER$2$INITIALIZATION: PROCEDURE PUBLIC;

                 /* INITIALIZE COUNTER 2 FOR EIGHT BIT COUNTING */
156    2             BELT$COUNT = 0 ;

                 /* RETURN TO CALLING PROGRAM */
157    2             RETURN;

158    2         END COUNTER$2$INITIALIZATION;
159    1         END PROCESSOR$INITIALIZATION$MODULE;
                 $EJECT
```

MODULE INFORMATION:

```
    CODE AREA SIZE      = 0201H     513D
    VARIABLE AREA SIZE = 0001H      1D
    MAXIMUM STACK SIZE = 0000H      0D
    329 LINES READ
    0 PROGRAM ERROR(S)
```

END OF PL/M-80 COMPILATION

```
                $INTVECTOR(4,3F00H)
                $PAGEWIDTH(72)
                $TITLE('PROCESSOR INTERFACE')
                /*********************************************
                * THESE PROGRAMS PROVIDE THE OPERATING INTER- *
                * FACE BETWEEN THE APPLICATION PROGRAM AND     *
                * THE ISBC 941 PROCESSORS OR COUNTERS.         *
                *********************************************/

  1             PROCESSOR$INTERFACE$MODULE: DO;

                /* DECLARATION OF ADDRESSES */
  2     1           DECLARE UPI$0$STATUS    LITERALLY 'ØE5H';
  3     1           DECLARE UPI$0$COMMAND   LITERALLY 'ØE5H';
  4     1           DECLARE UPI$0$DATA      LITERALLY 'ØE4H';

  5     1           DECLARE UPI$1$STATUS    LITERALLY 'ØE7H';
  6     1           DECLARE UPI$1$COMMAND   LITERALLY 'ØE7H';
  7     1           DECLARE UPI$1$DATA      LITERALLY 'ØE6H';

  8     1           DECLARE UPI$2$STATUS    LITERALLY 'ØE9H';
  9     1           DECLARE UPI$2$COMMAND   LITERALLY 'ØE9H';
 1Ø     1           DECLARE UPI$2$DATA      LITERALLY 'ØE8H';

                /* DECLARATION OF ISBC 941 COMMANDS */
 11     1           DECLARE SETP1           LITERALLY 'ØØBH';
 12     1           DECLARE CLRP1           LITERALLY 'ØØDH';
 13     1           DECLARE CLRP2           LITERALLY 'ØØEH';
 14     1           DECLARE RDFCØ           LITERALLY 'Ø42H';
 15     1           DECLARE RDFC1           LITERALLY 'Ø43H';
 16     1           DECLARE PAUSE           LITERALLY 'ØØ5H';
 17     1           DECLARE LOOP            LITERALLY 'ØØ4H';
 18     1           DECLARE INITPF          LITERALLY 'ØØ2H';
 19     1           DECLARE PACIFY          LITERALLY 'ØØ1H';
 2Ø     1           DECLARE ENFLAG          LITERALLY 'ØØ6H';
 21     1           DECLARE SETOE           LITERALLY 'Ø71H';

                /* DECLARATION OF ISBC 941 STATUS BITS */
 22     1           DECLARE RFC             LITERALLY 'Ø80H';
 23     1           DECLARE IBF             LITERALLY 'ØØ2H';
 24     1           DECLARE OBF             LITERALLY 'ØØ1H';
 25     1           DECLARE QF              LITERALLY 'Ø1ØH';
 26     1           DECLARE QNE             LITERALLY 'Ø2ØH';

                /* DEFINE THE MATH ROUTINES USED BY MODULES */
 27     1           MQULD4: PROCEDURE (IR$PTR,VALUE$PTR) EXTERNAL;
 28     2               DECLARE (IR$PTR,VALUE$PTR) ADDRESS;
 29     2               END MQULD4;
 3Ø     1           MQUDV2: PROCEDURE (IR$PTR,VALUE$PTR) EXTERNAL;
 31     2               DECLARE (IR$PTR,VALUE$PTR) ADDRESS;
 32     2               END MQUDV2;
```

```
33   1              MQUDV1: PROCEDURE (IR$PTR,VALUE$PTR) EXTERNAL;
34   2                  DECLARE (IR$PTR,VALUE$PTR) ADDRESS;
35   2                  END MQUDV1;
36   1              MQUST1: PROCEDURE (IR$PTR,VALUE$PTR) EXTERNAL;
37   2                  DECLARE (IR$PTR,VALUE$PTR) ADDRESS;
38   2                  END MQUST1;

            /* DEFINE THE MATH ACCUMULATOR STORAGE AREA */
39   1              DECLARE IR(18) BYTE EXTERNAL;


            /* DEFINE THE COUNTER LOCATIONS */
40   1              DECLARE (LIQ$COUNT,BELT$COUNT) BYTE EXTERNAL;


            $EJECT
            /**********************************************
            * THIS PROGRAM IS USED TO GENERATE A FREQUENCY   *
            * OUTPUT USING THE ISBC 941 MODULE INSTALLED IN  *
            * SOCKET NUMBER Ø.   TO PROVIDE MAXIMUM RESOLU-   *
            * TION, FOUR PERIODS WILL BE USED.   THE FREQUEN-*
            * CY RANGES CORRESPONDING TO EACH PERIOD ARE:    *
            *           RANGE        FREQ         RESOLUTION  *
            *             1       5Ø TO 165 HZ      2 HZ      *
            *             2      166 TO 225 HZ      3 HZ      *
            *             3      226 TO 285 HZ      3 HZ      *
            *             4      286 TO 55Ø HZ      6 HZ      *
            * THE SCALE FACTOR IS COMPUTED BY THE FORMULA:   *
            *     SF=1ØØØØØ/((FREQ)*(RANGE FACTOR))          *
            **********************************************/

41   1      WEIGHBELT$MOTOR$DRIVE: PROCEDURE (FREQ) PUBLIC;

            /* DECLARATION OF CONSTANT, 1ØØ,ØØØ */
42   2          DECLARE HUNDRED$K(4) BYTE DATA (
                    ØAØH,Ø86H,ØØ1H,ØØØH  );

            /* DECLARATION OF ISBC941 PORT ENABLES */
43   2          DECLARE ENABLE$FREQ   LITERALLY 'Ø1H';
44   2          DECLARE DISABLE$FREQ  LITERALLY 'ØØH';

            /* DECLARATION OF ISBC 941 MEMORY LOCATION COMMANDS */
45   2          DECLARE WRRM$55        LITERALLY 'Ø55H';
46   2          DECLARE WRRM$74        LITERALLY 'Ø74H';

            /* DECLARATION OF VARIABLES USED IN COMPUTATIONS */
47   2          DECLARE (RANGE,FREQA) BYTE;
48   2          DECLARE FREQ ADDRESS;

            /* BEGIN COMPUTATION OF OUTPUT FOR FREQ > 48 HZ. */
49   2          IF FREQ > 49
                THEN DO;

            /* ENABLE FREQUENCY OUTPUT */
51   3              DO WHILE ((INPUT(UPI$Ø$STATUS) AND IBF) <> Ø);
52   4              END;
53   3              OUTPUT(UPI$Ø$COMMAND) = SETOE;
```

```
54   3              DO WHILE ((INPUT(UPI$0$STATUS) AND IBF) <> 0);
55   4              END;
56   3              OUTPUT(UPI$0$DATA) = ENABLE$FREQ;


          /* COMPUTATION OF FREQUENCY RANGE */
57   3                  IF FREQ < 285
                        THEN DO;
59   4                      IF FREQ < 226
                            THEN DO;
61   5                          IF FREQ < 166
                                THEN RANGE = 9;
63   5                          ELSE RANGE = 5;
64   5                      END;
65   4                      ELSE RANGE = 3;

66   4                  END;
67   3                  ELSE RANGE = 2;


          /* LOAD MATH ACCUMULATOR WITH 100,000 */
68   3                  CALL MQULD4 (.IR,.HUNDRED$K);


          /* TEST FOR MOTOR SHUTDOWN */
69   3                  IF FREQ > 1
                        THEN DO;


          /* DIVIDE BY FREQUENCY */
71   4                      CALL MQUDV2 (.IR,.FREQ);


          /* DIVIDE BY RNAGE FACTOR */
72   4                      CALL MQUDV1 (.IR,.RANGE);


          /* GET TWO'S COMPLEMENT FOR ISBC 941 SCALE FACTOR */
73   4                      CALL MQUST1 (.IR,.FREQA);
74   4                      FREQA = NOT (FREQA + 1);
75   4                  END;


          /* ADJUST FOR MOTOR STOP SIGNAL */
76   3                  ELSE DO;
77   4                      FREQA = 000H;
78   4                      RANGE = 0FFH;
79   4                  END;


          /* SEND NEW SCALE FACTOR TO DEVICE */
80   3                  DO WHILE ((INPUT(UPI$0$STATUS) AND IBF) <> 0);
81   4                  END;
82   3                  OUTPUT(UPI$0$COMMAND) = WRRM$55;

83   3                  DO WHILE ((INPUT(UPI$0$STATUS) AND IBF) <> 0);
84   4                  END;
85   3                  OUTPUT(UPI$0$DATA) = FREQA;


          /* SEND NEW PERIOD TO DEVICE */
86   3                  DO WHILE ((INPUT(UPI$0$STATUS) AND IBF) <> 0);
87   4                  END;
88   3                  OUTPUT(UPI$0$COMMAND) = WRRM$74;
```

```
89    3              DO WHILE ((INPUT(UPI$0$STATUS) AND IBF) <> 0);
90    4              END;
91    3              OUTPUT(UPI$0$DATA) = RANGE;

      /* END OF FREQUENCY OUTPUT MODE */
92    3          END;

      /* HANDLE FREQUENCIES < 50 HZ. */
93    2          ELSE DO;

      /* DISABLE FREQUENCY OUTPUT GENERATION */
94    3              DO WHILE ((INPUT(UPI$0$STATUS) AND IBF) <> 0);
95    4              END;
96    3              OUTPUT(UPI$0$COMMAND) = SETOE;

97    3              DO WHILE ((INPUT(UPI$0$STATUS) AND IBF) <> 0);

98    4              END;
99    3              OUTPUT(UPI$0$DATA) = DISABLE$FREQ;

      /* END OF ALTERNATE FREQUENCY OUTPUT */
100   3          END;

      /* RETURN TO CALLING PROGRAM */
101   2          RETURN;

102   2      END WEIGHBELT$MOTOR$DRIVE;

      $EJECT
      /**********************************************************
      * THIS PROGRAM GETS  THE WEIGHBELT WEIGHT FROM THE        *
      * NUMBER 1 ISBC 941 PROCESSOR.  THE WEIGHT WILL BE        *
      * RECEIVED AS A COUNT WHICH RANGES BETWEEN 0 AND          *
      * 2000, CORRESPONDING TO A WEIGHT BETWEEN 0.0 AND         *
      * 10.00 POUNDS.  EACH COUNT RECEIVED  HAS A VALUE         *
      * OF 0.005 POUNDS.                                        *
      **********************************************************/

103   1      WEIGHBELT$WEIGHT: PROCEDURE ADDRESS PUBLIC;

      /* DECLARATIONS OF VARIABLES USED IN THE PROCEDURE */
104   2          DECLARE (LCOUNT,HCOUNT) BYTE;
105   2          DECLARE WEIGHT ADDRESS;

      /* GET INPUT COUNT LOW BYTE */
106   2          DO WHILE ((INPUT(UPI$1$STATUS) AND IBF) <> 0);
107   3          END;
108   2          OUTPUT(UPI$1$COMMAND) = RDFC0;

109   2          DO WHILE ((INPUT(UPI$1$STATUS) AND OBF) = 0);
110   3          END;
111   2          LCOUNT = INPUT(UPI$1$DATA);
```

```
                 /* GET INPUT COUNT HIGH BYTE */
112    2             DO WHILE ((INPUT(UPI$1$STATUS) AND IBF) <> Ø);
113    3             END;
114    2             OUTPUT(UPI$1$COMMAND) = RDFC1;

115    2             DO WHILE ((INPUT(UPI$1$STATUS) AND OBF) = Ø);
116    3             END;
117    2             HCOUNT = INPUT(UPI$1$DATA);

                 /* START NEXT WEIGHT SAMPLE PERIOD */
118    2             DO WHILE ((INPUT(UPI$1$STATUS) AND IBF) <> Ø);
119    3             END;
120    2             OUTPUT(UPI$1$COMMAND) = LOOP;

                 /* CONVERT WEIGHT TO AN ADDRESS VALUE */
121    2             WEIGHT = HCOUNT;
122    2             WEIGHT = SHL(WEIGHT,8);
123    2             WEIGHT = WEIGHT + LCOUNT;

                 /* DIVIDE BY TWO TO CONVERT TO POUNDS */
124    2             WEIGHT = SHR(WEIGHT,1);

                 /* RETURN THE WEIGHTBELT WEIGHT */
125    2             RETURN WEIGHT;

126    2         END WEIGHBELT$WEIGHT;
                 $EJECT
                 /***************************************************
                 * THIS PROCEDURE TRANFERS THE WEIGHBELT SPEED TO   *
                 * THE CALLING PROGRAM AND CLEARS THE COUNTER FOR    *
                 * THE NEXT TEST.   THE SPEED RESOLUTION PROVIDES    *
                 * ONLY FIVE SPEED RANGES.                           *
                 ***************************************************/

127    1         WEIGHBELT$SPEED: PROCEDURE BYTE PUBLIC;

                 /* DECLARATIONS OF VARIABLES USED BY THE PROCEDURE */
128    2             DECLARE SPEED BYTE;

                 /* LATCH COUNTER BEFORE READING SPEED */
129    2             DISABLE;

                 /* GET COUNTER VALUE FROM COUNTER */
130    2             SPEED = BELT$COUNT;

                 /* CLEAR COUNTER FOR NEXT OPERATION */
131    2             BELT$COUNT = Ø;
132    2             ENABLE;

                 /* RETURN DATA TO CALLING ROUTINE */
133    2             RETURN SPEED;

134    2         END WEIGHBELT$SPEED;
```

```
$EJECT
/****************************************************
* THIS PROCEDURE PROVIDES COMMANDS TO THE STEPPER   *
* MOTOR TO OPERATE THE CONTROL VALVE.  IT WILL COM-*
* PUTE THE SIGNED MAGNITUDE REPRESENTATION FROM     *
* THE TWO'S COMPLIMENT INPUT AND WILL ISSUE THE     *
* APPROPRIATE STEP INCREMENT AND DIRECTION.    A    *
* FIXED STEP RATE OF 100 STEPS PER SECOND WILL BE   *
* USED BY THE CONTROL DEVICE.                       *
****************************************************/
```

135   1   LIQUID$VALVE$POSITION: PROCEDURE (POSITION) PUBLIC;

```
                  /* DECLARATIONS OF VARIABLES USED BY THE PROCEDURE */
136    2              DECLARE POSITION BYTE;

                  /* DEFINITIONS OF TERMS USED IN COMPUTATIONS */
137    2              DECLARE STEP$RATE LITERALLY '005H';
138    2              DECLARE REVERSE    LITERALLY '080H';

                  /* IF NO MOVEMENT, SKIP OPERATIONS */
139    2              IF POSITION <> 0
                      THEN DO;

                  /* SUPPORT CONVERSION TO SIGNED MAGNITUDE NUMBER */
141    3                  IF POSITION > 127
                          THEN DO;

                  /* GET MAGNITUDE OF MOVEMENT */
143    4                      POSITION = 256 - POSITION;

                  /* SET SIGN FOR CCW ROTATION */
144    4                      POSITION = POSITION OR REVERSE;
145    4                  END;

                  /* VERIFY THAT QUEUE SPACE IS AVAILABLE */
146    3                  DO WHILE ((INPUT(UPI$2$STATUS) AND QF) <> 0);
147    4                  END;

                  /* REQUEST DESIRED STEP RATE */
148    3                  DO WHILE ((INPUT(UPI$2$STATUS) AND IBF) <> 0);
149    4                  END;
150    3                  OUTPUT(UPI$2$DATA) = STEP$RATE;

                  /* REQUEST STEPPER MOVEMENT */
151    3                  DO WHILE ((INPUT(UPI$2$STATUS) AND IBF) <> 0);
152    4                  END;
153    3                  OUTPUT(UPI$2$DATA) = POSITION;
154    3              END;

                  /* RETURN TO CALLING PROGRAM */
155    2              RETURN;

156    2          END LIQUID$VALVE$POSITION;
```

```
                $EJECT
                /****************************************************
                * THIS PROCEDURE TRANSFERS THE LIQUID FLOW RATE FROM *
                * THE FLOW COUNTER TO THE CALLING PROGRAM.   AFTER    *
                * READING, THE FLOW COUNTER WILL BE RESET TO FACILI-  *
                * TATE THE NEXT READING.  THE LIQUID FLOW COUNT WILL  *
                * VARY BETWEEN 20 AND 240 PULSES IN EACH 200 MILLI-   *
                * SECOND SAMPLE INTERVAL.  THIS WILL CORRESPOND TO    *
                * THE ACTUAL LIQUID FLOW RATE OF 10 TO 120 POUNDS     *
                * PER MINUTE.                                         *
                ****************************************************/

157    1        LIQUID$FLOW$RATE: PROCEDURE ADDRESS PUBLIC;

                /* DECLARATION OF VARIABLES USED BY THE PROGRAM */
158    2            DECLARE TEMP BYTE;
159    2            DECLARE (FLOW,T$TWO,T$SXTN,T$THRTWO) ADDRESS;

                /* LATCH COUNTER BEFORE READING FLOW */
160    2            DISABLE;

                /* GET FLOW RATE VALUE FROM COUNTER */
161    2            TEMP = LIQ$COUNT;

                /* CLEAR COUNTER FOR NEXT OPERATION */
162    2            LIQ$COUNT = 0;
163    2            ENABLE;

                /* CONVERT TO POUNDS PER MINUTE */
164    2            FLOW = TEMP;
165    2            T$TWO = SHL(FLOW,1);
166    2            T$SXTN = SHL(T$TWO,3);
167    2            T$THRTWO = SHL(T$SXTN,1);
168    2            FLOW = T$TWO + T$SXTN + T$THRTWO;

                /* RETURN FLOW RATE TO CALLING PROGRAM */
169    2            RETURN FLOW;

170    2        END LIQUID$FLOW$RATE;

                $EJECT
                /*******************************************
                *   COUNTER FOR LIQUID FLOW RATE FROM LIQUID *
                * FLOW METER.  COUNT PULSE WILL GENERATE AN  *
                * INTERRUPT AT LEVEL 1.                      *
                *******************************************/

171    1        LIQ$CNT: PROCEDURE INTERRUPT 1 PUBLIC;

                /* INCREMENT FLOW COUNT */
172    2            LIQ$COUNT = LIQ$COUNT + 1;

                /* SEND END OF INTERRUPT */
173    2            OUTPUT (0ECH) = 020H;
```

```
                    /* RETURN */
 174    2               RETURN;

 175    2           END LIQ$CNT;

                    $EJECT
                    /*******************************************
                    * THIS PROCEDURE WILL PROVIDE AN EVENT COUN-*
                    * TER TO HANDLE THE BELT MOTION DETECTOR.   *
                    * IT WILL OPERATE BY DIRECTING THE MOTION   *
                    * PULSE TO INTERRUPT 2.                     *
                    *******************************************/

 176    1           BELT$CNT: PROCEDURE INTERRUPT 0 PUBLIC;

                    /* INCREMENT BELT MOVEMENT */
 177    2               BELT$COUNT = BELT$COUNT + 1;

                    /* SEND END OF INTERRUPT */
 178    2               OUTPUT (0ECH) = 020H;

                    /* RETURN */
 179    2               RETURN;

 180    2           END BELT$CNT;
 181    1           END PROCESSOR$INTERFACE$MODULE;



MODULE INFORMATION:

    CODE AREA SIZE     = 0251H     593D
    VARIABLE AREA SIZE = 0013H      19D
    MAXIMUM STACK SIZE = 0008H       8D
    400 LINES READ
    0 PROGRAM ERROR(S)

END OF PL/M-80 COMPILATION
```

Designing and Assembling Microcomputer Systems Grows Easier

By Michael J. McGowan
Control Engineering/February 1979

Designing and Assembling Microcomputer Systems Grows Easier

Although a single data bus standard yet eludes the microcomputer industry, numerous manufacturers of single-board computer and supplementary boards have cast a hardware vote for the Multibus, Intel's microcomputer backplane which they originated in 1976. With a steady eye on the control industry market, Intel has designed a home to accommodate Multibus compatible equipment, the iCS-80 industrial chassis. It promises to significantly reduce the time and cost of assembling the housing and interface parts of a microcomputer-based control system. In this article, besides taking the first look at Intel's new chassis and signal conditioning panels, we've put together a comprehensive list of Multibus compatible equipment.

MICHAEL J. McGOWAN, Control Engineering

After the development of single-board computers nearly three years ago, vendors moved quickly to seize a fraction of the market. It seemed at first that everything from memories to analog I/O boards had become available. With an astonishing suddenness, companies sprang up in Silicon Valley, Texas, New Jersey, and along the forested roadside of Rt. 128 outside of Boston. Late that year, we counted well over a hundred companies anxious to make their fortune selling the control engineer everything from one or two interface boards to complete microprocessor systems.

Then the pleasant dream became a nightmare. From power supply requirements to backplane pinouts, little was compatible. Even such an obvious thing as board size differed from vendor to vendor and many a hope for an ideal system was crushed in a pragmatic search for whatever would fit together.

Seven months ago in our June 1978 issue we noted that the number of microcomputer system manufacturers had dwindled to about 60 and since then, we find still fewer. Some, no doubt, were forced out for lack of reliability, though most, despite remarkably talented engineering, starved as the market saturated.

Large scale integration of microcomputer components has more than doubled the memory size of single-board computers. Sixteen bit word lengths will become commonplace in the next year as microcomputer performance begins to rival the minicomputer's and the lines of distinction between micros and minis fades.

The fight for a standard data bus drags on with leaders in the struggle but no winner. On the offensive, Pro-Log and Mostek jointly introduced the STD bus last autumn in an attempt to gain a greater market share by espousing decentralized system architectures. Their philosophy argues economics: the user should pay only for essential functions by selecting small, specialized boards
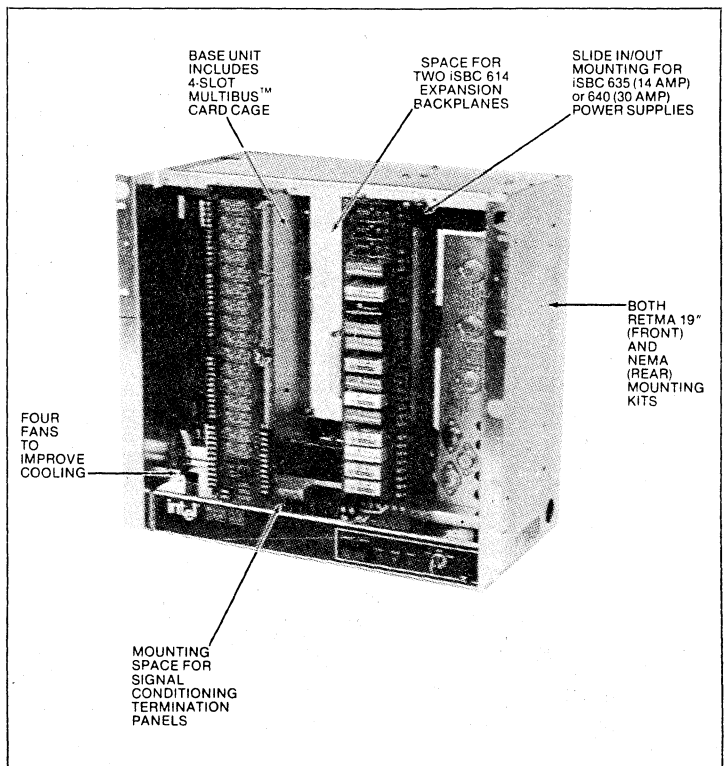
and not squander funds on a general purpose board with extra features.

Still, Intel, favoring more densely packed and versatile boards, continues to dominate the market while the Multibus retains its popularity. Today more 30 manufacturers produce over one hundred different boards based on that bus structure alone. Not that Intel enjoys the strict fidelity of its outside vendors; Digital Equipment Corporation, for instance, boasts some 17 companies providing boards to mate with the LSI-11 and LSI-112.

But perhaps alone among its competitors, Intel has recognized that the majority of its boards are being used in industrial applications and that the control system designer needs more than components.

An industrial chassis

A microcomputer system designer must choose components that are electro-mechanically compatible. To that end, Intel is introducing the iCS-80 industrial chassis and termination panels. It makes all Multibus-compatible CPU

BASE UNIT INCLUDES 4-SLOT MULTIBUS™ CARD CAGE

SPACE FOR TWO iSBC 614 EXPANSION BACKPLANES

SLIDE IN/OUT MOUNTING FOR iSBC 635 (14 AMP) or 640 (30 AMP) POWER SUPPLIES

BOTH RETMA 19″ (FRONT) AND NEMA (REAR) MOUNTING KITS

FOUR FANS TO IMPROVE COOLING

MOUNTING SPACE FOR SIGNAL CONDITIONING TERMINATION PANELS

MULTIBUS Compatible Boards and Vendors	Analog Input and Output Boards	Analog Signal Conditioning/Screw Terminations	Communications Boards	Core Memory Boards	Digital Input/Output Boards	Digital Relay Boards	CPU Boards	DMA Controller Boards	Floppy Disk Controller Boards	Hard Disk Interface Boards	IEEE-488 Bus Interface Boards	Keyboard Controller Boards	Math Boards	Optical Isolator Boards (without screw terminations)	Optical Isolation With Screw Terminations	PROM Boards	RAM Boards	RAM/PROM or ROM	Synchro to Digital Boards	Tape (cartridge/cassette) Interface Boards	Tape (½") Interface Boards	Video Boards	Wire Wrap Boards
ADAC Corp.	●																						
Advanced Micro Computers							●							●									
Ampex				●																			
Analog Devices	●																						
Augat																							●
Burr-Brown	●				●																		
Computer Marketing	●								●	●							●				●	●	●
Data Translation	●																						
Datacube																●	●	●				●	
Datel Systems	●																						
Electronic Solutions																●	●						
Garry Manufacturing																							●
HT Instruments																				●			
Hal Communications																						●	
Heurikon	●						●		●								●	●				●	●
IDEAS	●								●		●		●						●				
Intel	●	●	●	●	●	●			●		●	●	●		●	●	●	●					●
Interphase			●						●	●												●	
Matrox Electronic Systems																						●	
Megalogic																●	●	●		●			
Micro Memories																							
Micro Networks	●			●																			
MicroTec																●							
Micro/Tel									●														
Monolithic Systems							●		●									●					●
Motorola																	●						
MUPRO																	●						
National Semiconductor	●	●	●		●	●			●			●	●			●	●	●				●	●
North Star Computers													●										
Pertec (ICOM)									●														
Relational Memory Systems							●																
Systems, Computers and Interfaces							●		●				●				●	●					●
Thomas Engineering Co.																							
Vector Electronic																							●
XEDAX										●											●		
ZIA Tech										●													

and peripheral boards readily usable.

The advantage of the ICS-80 is that most of the interconnection and mechanical details for assembling a microcomputer-based control system have already been worked out.

The iCS-80 stands 15.75 inches high and can be mounted in a stardard RETMA 19 inch rack, or in a NEMA cabinet secure from the industrial environment. The minimum layout consists of a four-slot Multibus card cage with provisions for adding two more cages to a maximum of twelve cards. The cages fit vertically, like records in a rack, to aid convection cooling and permit front access for insertion and maintenance.

On the right side of the chassis is room for either a 14 or 30 ampere power supply, the choice dictated by the application. The system will operate on either 115 or 230 volts with a range of 47 to 63 Hertz specified in anticipation of international service.

Cooling is assisted by four fans—three for the card cages and one for the power supply section. The intention here is to make the installation of additional fans unnecessary even after the system has expanded. The fans are expected to provide adequate cooling for most applications so supplementary air conditioning can be eliminated or at least minimized.

Signal conditioning

Three signal conditioning panels have been developed by Intel to simplify connections between the processing cards and the outside world. The principle is neatness, and with that follows reliability. Flat ribbon cables connect the signal conditioners to the processor cards, a safeguard from "which wire is which" and screwdriver slips in the vicinity of expensive boards. Field connections to the external inputs and outputs are made (presumably by electricians with big hands and reputations for being less than delicate) through rugged, screw-type barrier strips that accept wire as heavy as 14 AWG. The panels can mount either on RETMA cabinet brackets, NEMA wall spacers, or on the iCS-80 chassis itself.

Each signal conditioning card gives the user a variety of options. The iCS-910 analog signal conditioning/termination panel accepts up to 16 differential or 32 single ended input channels. The four 2-wire analog output channels might be connected to 4 to 20 mA current loops.

The digital signal conditioning termination panel, iCS-920, handles 24 two-wire input or output channels with signals up to 55 V, 300 mA. Inputs can be diode protected, and pads are provided for current limiters or voltage dividers. Optoisolators may be inserted in

the DIP sockets for high voltage isolation or jumpers may be used instead when the input is TTL. Similarly, output sockets accept jumpers for direct TTL output, DIP optoisolators for transient suppression, or integrated circuit (open collector) drivers for high voltage to high current outputs. Activity on each channel is indicated by LEDs.

The ac signal conditioning/(solidas) termination panel, iCS-930, will actually work with ac or dc on its 16 channels. The user supplies optoisolators for input isolation and optically-isolated solid

state relays for output isolation. Mounting pads for customer-supplied MOVs or snubber networks are included. As before, a fuse gives overload protection and LEDs indicate channel activity.

The advantage of all this is that by plugging in some components and perhaps inserting a few resistors and capacitors, the interface units can be tailored to a particular application. Since many mechanical and electrical connection problems have already been solved, a customized unit can be built with minimum effort. □

Vendors of Multicompatible Boards

ADAC Corp.
Woburn, MA
617/935-6668

Advanced Micro Computers
Santa Clara, CA
408/732-2400

Ampex
El Segundo, CA
714/973-2970

Analog Devices
Norwood, MA
617/329-4700

Augat
Attleboro, MA
617/222-2202

Burr-Brown
Tucson, AZ
602/655-8000

Computer Marketing
Waltham, MA
617/894-7000

Data Translation
Natick, MA
617/655-5300

Datacube
Reading, MA
617/944-4600

Datel Systems
Canton, MA
617/828-8000

Electronic Solutions
San Diego, CA
714/292-0242

Garry Manufacturing
New Brunswick, NJ
212/267-6844

HT Instruments
Marina Del Rey, CA
312/822-4296

Hal Communications
Urbana, IL
217/367-7373

Heurikon
Madison, WI
608/255-9075

IDEAS
Beltsville, MD
301/937-3600

Intel
Aloha, OR
503/642-2563

Interphase
Dallas, TX
214/238-0971

Matrox Electronic Systems
Montreal, Quebec
514/735-1182

Megalogic
Brookville, OH
513/833-5222

Micro Memories
Chatsworth, CA
213/998-0070

Micro Networks
Worcester, MA
617/852-5400

MicroTec
Sunnyvale, CA
408/733-2919

Micro/Tel
St. Louis, MO
314/569-3450

Monolithic Systems
Englewood, CO
303/770-7400

Motorola
Austin, TX
512/928-6572

MUPRO
Sunnyvale, CA
408/737-0500

National Semiconductor
Santa Clara, CA
408/737-5262

North Star Computers
Berkeley, CA
415/549-0858

Pertec (ICOM)
Chatsworth, CA
213/998-1800

Relational Memory Systems
San Jose, CA
408/248-6356

Systems, Computers and Interfaces
Waltham, MA
617/899-2359

Thomas Engineering Co.
Concord, CA
415/686-3041

Vector Electronic
Sylmar, CA
213/365-9661

XEDAX
Alameda, CA
415/521-6600

ZIA Tech
Cupertino, CA
408/996-7082

DOCUMENTATION

RELATED INTEL PUBLICATIONS

System 80/10 Microcomputer Hardware Reference Manual, 98-00316B

iSBC 80/10 and iSBC 80/10A Single Board Computer Hardware Reference Manual, 9800230F

iSBC 80P and iSBC 80P10 Prototyping Package User's Guide, 9800223D

iSBC 80/20 and iSBC 80/20-4 Single Board Computer Hardware Reference Manual, 98-317C

iSBC 80/30 Hardware Reference Manual, 9800611A

iSBC 86/12 Single Board Computer Hardware Reference Manual, 9800645A

iSBC 544 Intelligent Communications Controller Board Hardware Reference Manual, 9800616B

iSBC 569 Intelligent Digital Controller Board Hardware Reference Manual, 9800845

iSBC 941 Industrial Digital Processor User's Guide, 9803077-02

iCS 80 Industrial Chassis Hardware Reference Manual, 9800799A

iSBC 310 High Speed Mathematics Unit Hardware Reference Manual, 9800410A

iSBC 957 Intellec–iSBC 86/12 Interface and Execution Package User's Guide, 9800743A

Intel MULTIBUS Specification, 9800683

MCS-80 User's Manual, 98-153D

MCS-85 User's Manual, 98-366C

The 8086 Family User's Manual

UPI-41 User's Manual, 9800504

Introduction to the UPI-41A, AP-41

RMX/80 User's Guide, 9800522C

ISIS-II User's Guide, 9800306D

8080/8085 Assembly Language Programming Manual, 9800301C

PL/M-80 Programming Manual, 9800268B

ISIS-II PL/M-80 Compiler Operator's Manual, 9800300

FORTRAN-80 Programming Manual, 9800481A

ISIS-II FORTRAN-80 Compiler Operator's Manual, 9800480B

"How to use FORTRAN with other Intel Languages", AP-44

BASIC-80 Reference Manual, 9800758

A Guide to Intellec Microcomputer Development Systems by Daniel D. McCracken, 9800558B

8080/8085 Fundamental Support Package (FSP) Reference and Operating Instructions for ISIS-II Users, 9800887-01

8086 Assembly Language Reference Manual, 9800640A

MCS-86 Assembler Operator's Instructions for ISIS-II Users, 9800641A

PL/M-86 Programming Manual, 9800466A

ISIS-II PL/M-86 Compiler Operator's Manual, 9800478A

ISIS-II 8086 Cross Development Utilities Operator's Manual, 9800639A

TECHNICAL LITERATURE LIST

Title	Part No.

MEMORY COMPONENTS

Memory Design Handbook — 1979	011100
Growing Static RAM Family Album	010100
2115A/2125A Brochure	001710
RR 7 — 2107A/2107B Reliability	006540
RR 8 — Polysilicon fuse Bipolar PROM	006560
RR 11 — 2416 16K CCD Memory	006700
RR 12 — 2708 8K Erasable PROM	006720
RR 14 — 2115/2125 MOS Static RAMS	006740
RR 15 — 2104A	006750
RR 16 — 2116	006760
RR 18 — HMOS Reliability Update	006771
RR 19-2716 — UV Erasable PROM	006775
RR 20-2117 — Reliabilty	006780
AR 20 — 16K RAM	006900
AR 35-2716 — Erasable PROM–16,384 Bits On-Chip	007300
AR 44 — Speedy RAM Runs Cool — 2147	007320
AR 46 — HMOS Scales Traditional Devices	007330
AR 78 — ISSCC Reprint on Static RAMS	007370
AP 22 — Which Way for 16K	008300
AP 23 — 2104A 4K RAM	008500
AP 30 — Applications of 5 Volt EPROM & ROM Family	008550
AP 46 — Error Detecting and Correcting Codes	008560

TELECOM

AR 79 — ISSCC Reprint — 2920	007375
AR 80 — ISSCC Reprint — 2912	007380
AR 81 — Single Chip NMOS Micro-process Signals	007385
AR 88 — First Monolithic PCM Filter	007400

MAGNETICS

Bubble Memory Design Handbook	900020
AR 92 — Megabit Bubble Memory Chip Gets Support from LSI	900500
AR 96 — Here Comes A Million Bit Chip	900515
A Total System Solution to Magnetics Applications (Technical Paper)	900520

MICROCOMPUTER COMPONENTS

MCS 48 User's Manual	98-270
MCS 48 Product Description (98-615)	201710
MCS 48 Applications Handbook	121511
MCS 48 Reference Card (98-412)	202300
AP 24 — MCS 48 Family (98-413)	203800
AP 40 — Keyboard/Display Scanning . . . MCS 48 (98-755)	203805
AP 49 — Serial I/O and Math Utilities . . . 8049 (98-904)	203810
AP 55A — High Speed Emulator for MCS 48	203815
AP 56 — Designing With Intel's 8022 Micro (98-954)	203820
AR 58 — Microcontroller Includes A-D Converter (98-718)	203605
AR 63 — Microcomputer's On-Chip Functions — 8022 (98-780)	203610
AR 102 — Designing Reliable Software for Auto Applications	207350
AR 107 — Use EPROM 1-Chip μCs as Effective 1-Shot Lab Aids	207355
UPI-41 User's Manual	98-504
UPI-41 Reference Card (98-671)	203100
MCS-48 and UPI-41 Assembly Language Programming Manual	98-255
MCS-80 User's Manual	98-153
RR 10 — 8080/8080A Microcomputer	207100
MCS-85 User's Manual	98-366
MCS-85 Product Description (98-365)	205770
8080/8085 Reference Card (98-438)	205785
AP 29 — Using the Intel 8085 Serial I/O Lines (98-684)	207715
8080/8085 Assembly Language Programming Manual	98-940
8080/8085 Floating Point Arithmetic Library User's Manual	98-452

Title	Part No.
AP 26 — iSBC 80/10–System 80/10	511000
RR 17 — iSBC 80/10 Reliability	509000
iSBC 80/20 and iSBC 80/20A Hardware Reference Manual	98-317
AR 28 — Control Engineering iSBC 80/20 Description	510100
iSBC 80/30 Hardware Reference Manual	98-611
AR 65 — Triple Bus Architecture (iSBC 80/30)	510140
iSBC 957 Intellec iSBC 86/12 User's Guide	98-743
AP 43 — Using the iSBC 957 (98-816)	511030
iSBC 86/12 Hardware Reference Manual	98-3075
AR 72 — 16-Bit Single Board Computer	510160
AR 69 — Dual-Port RAM Hikes Throughput (iSBC 80/30)	510150
iSBC 016 16K RAM Expansion Board Hardware Reference Manual	98-279
iSBC 032/048/064 Random Access Memory Boards Hardware Reference Manual	98-488
iSBC 094 4K-Byte CMOS RAM/Battery Backup Board Hardware Reference Manual	98-449
iSBC 104/108/116 Combination Memory and I/O Expansion Boards Hardware Reference Manual	98-277
iSBC 202 Double Density Diskette Controller Hardware Reference Manual	98-420
iSBC 204 Flexible Disk Hardware Reference Manual	98-568
iSBC 206 Disk Controller Hardware Reference Manual	98-567
iSBC 310 High-Speed Mathematics Unit Hardware Reference Manual	98-410
iSBC 416 16K PROM/ROM Expansion Board Hardware Reference Manual	98-265
iSBC 464 PROM/ROM Board Hardware Reference Manual	98-643
iSBC 501 Direct Memory Access Controller Hardware Reference Manual	98-294
iSBC 508 I/O Expansion Board Hardware Reference Manual	98-278
iSBC 517 Combination I/O Expansion Board Hardware Reference Manual	98-388
iSBC 519 Programmable I/O Expansion Board Hardware Reference Manual	98-385
iSBC 534 Four-Port Communications Expansion Board Hardware Reference Manual	98-450
iSBC 544 Intelligent Communications Controller Board Hardware Reference Manual	98-616
iSBC 556 Optically Isolated Programmable I/O Board Hardware Reference Manual	98-489
iSBC 569 Intelligent Digital Controller Hardware Reference Manual	98-845
iSBC 604/614 Cardcage Hardware Reference Manual	98-708
iSBC 635 Power Supply User's Manual	98-298
iSBC 640 Power Supply Hardware Reference Manual	98-803
iSBC 660 System Chassis Hardware Reference Manual	98-505
iSBC 915 GO-NO-GO Diskette Diagnostic and Monitor Program User's Manual	98-350
System 80/10 Microcomputer Hardware Reference Manual	98-316
System 80/20-4 Microcomputer Hardware Reference Manual	98-484
System 80/30 User's Guide	98-710
AR 48 — Reduce your Micro-based system design time	510110
AR 55 — Design Motivations for Multiple Processor Micro Systems	510120
AR 64 — Microcomputers — Single Chip or Single Board	510130
AP 28A — MULTIBUS Interfacing (98-587)	511010
Intel Delivers 8-bit/16-bit BM Configuration Envelopes	501100

INTELLEC MICROCOMPUTER DEVELOPMENT SYSTEM

Intellec 800 Operator's Manual	98-129
Intellec Reference Manual	98-132
Intellec Diagnostic Confidence Test Operator's Manual	98-386
Intellec Double Density DOS Hardware Reference Manual	98-422
ISIS I DOS Operator's Manual	98-206
Diskette Operating System Manual	98-212
Paper Tape Reader Guide	98-016
Series II Hardware Reference Manual	98-556
Series II Model 210 User's Guide	98-557
Intellec Series II Functional Description and Specifications (98-606)	404010
Intellec Series II Installation and Service Manual	98-559
Intellec Series Hardware Interface Manual	98-555
Success Manual for Single-Chip Microcomputer Users	402050
Success Manual for 8086 Users	402100
Microcomputer Development Package Booklet	404000
AR 97 — Minimizing Risk Through Use of Micro Development Systems	451130

Title	Part No.

SOFTWARE

iCIS Cobol Language Reference Manual	98-927
iCIS Cobol Packet Reference Card (98-929)	409100
2920 Assembly Language Manual	98-987
2920 Simulator User's Guide	98-988
FORTRAN-80 Programming Manual	98-481
FORTRAN-80 Reference Card (98-547)	400600
AR 73 — 8080 gets a "full blown" FORTRAN (98-844)	451125
PL/M Programming Manual	98-268
AR 59 — Modular Programming in PL/M	451115
PL/M 86 Programming Manual	98-466
BASIC-80 Reference Manual	98-758
BASIC-80 Reference Guide (98-774)	400705
AR 61 — Microprocessor Software Development Tools	451120
AR 98 — Software Development Package for 8086 System Designers	451135
ISIS II FORTRAN-80 Compiler Operator's Manual	98-480
ISIS II PL/M Compiler Operator's Manual	98-300
ISIS II PL/M 86 Compiler Operator's Manual	98-478
ISIS II 8085 Macro Assembler Operator's Manual	98-292
ISIS II System User's Guide	98-306
ISIS II Reference Card (98-841)	403350
ISIS II CREDIT User's Guide	98-902
CREDIT CRT-Based Text Editor Pocket Reference (98-903)	407700
MCS-86 Assembly Language Converter Operating Instructions for ISIS II Users	98-642
MCS-86 Assembly Operation Instructions for ISIS II Users	98-641
MCS-86 Software Development Utilities Operating Instructions for ISIS II Users	98-639
ICE-86 Operating Instructions for ISIS II Users	98-714
ICE-49 Operating Instructions for ISIS II Users	98-632
Multi-ICE Operating Instructions for ISIS II Users	98-672
iCIS-COBOL Compiler Operator's Instructions for ISIS II Users	98-928
8089 Assembler User's Manual	98-938

EMULATORS

ICE-30 Reference Manual	98-220
ICE-41 Operator's Manual	98-465
ICE-41 Reference Card (98-766)	305075
ICE-48 Operator's Manual	98-464
MCS-48 ICE Reference Card (98-653)	303925
ICE-80 Reference Manual	98-167
ICE-80 Operator's Manual	98-185
ICE-85 Operating Instructions	98-463
ICE-85 Brochure	406215
ICE-86 Pocket Reference (98-838)	406310
Multi-ICE Reference Card (98-810)	406505

PROM PROGRAMMERS

Universal PROM Programmer User's Manual	98-819
Universal PROM Programmer Reference Manual	98-133

PROMPT

PROMPT 48 Microcomputer User's Manual	98-402
PROMPT 48 Reference Card (98-404)	304850
PROMPT 80/85 User's Manual	98-307

μSCOPE

μScope 820 Operator's Handbook	98-526
μScope Reference Card (98-582)	408150
μScope 8080A Probe Service Manual	98-592
μScope 8085 Probe Service Manual	98-728
μScope Console Service Manual	98-593

Title	Part No.
μScope 820 Micro-Console Key Sequence Guide	98-826
AP 42 — Writing Diagnostics for the μScope (98-753)	452005

ADD-IN/ADD-ON MEMORY SYSTEMS

in-7000/in-7001 Product Description	888200
in-1670 Product Description	888210
in-4011 Product Description	888220
in-5034 Product Description	888230
Series 90 Product Description — CM90	888240
Series 90 Product Description — CM92	888250
Series 90 Configuration Guide	888790
AP 63 — Control and Interleaving BXP Standard Memory Bus	888510