November 1979

Closed Loop Control Using The iSBC 569/941™ Intelligent Digital Processors

Peter Andersen
OEM Microcomputer Systems Applications

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.

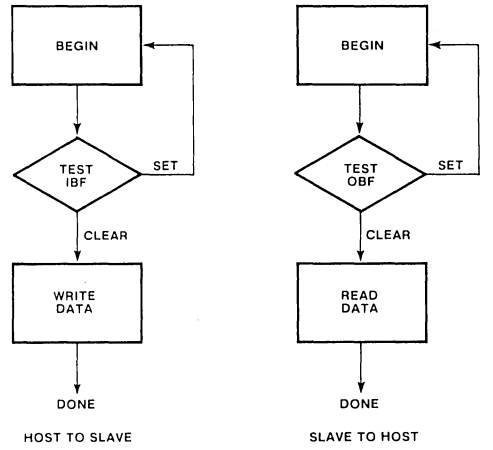


Figure 1. Simple Protocol

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

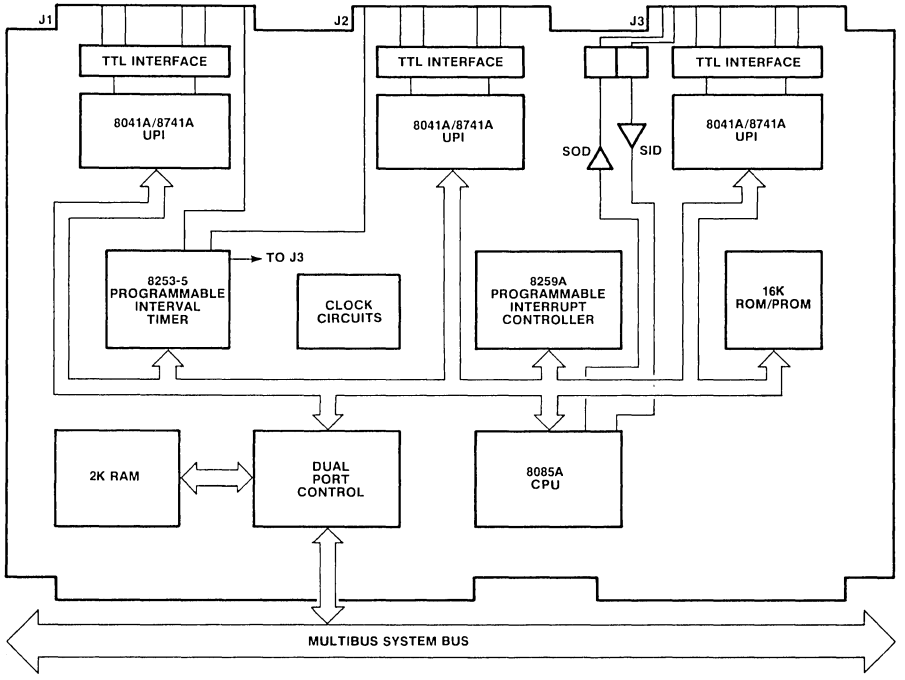


Figure 2. IDC Functional Block Diagram

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

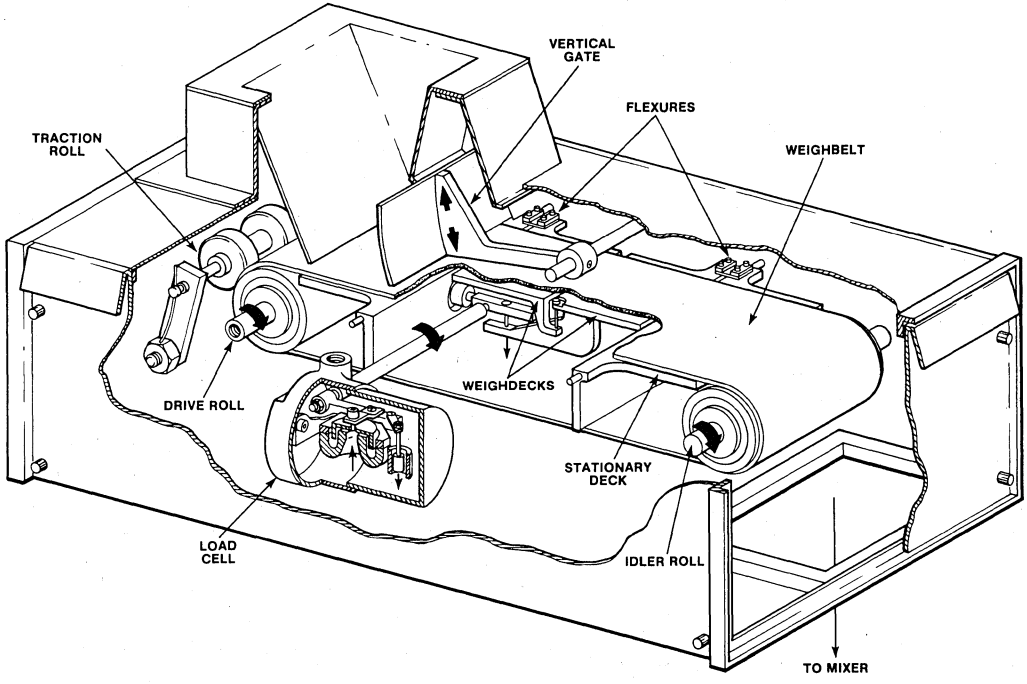


Figure 3. A Weighbelt Feeder

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.

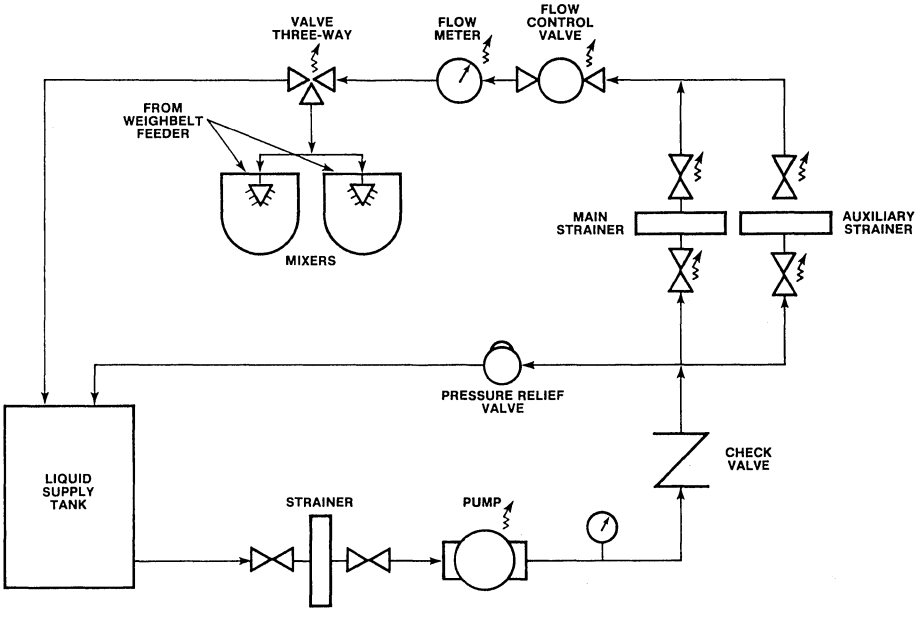


Figure 4. Liquid Flow Diagram

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.

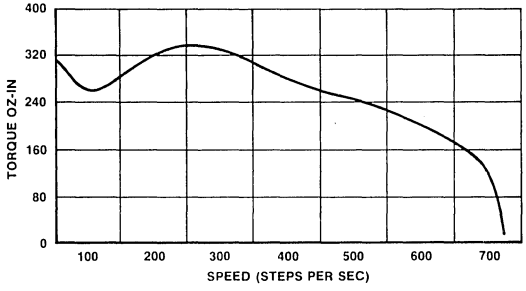


Figure 5. Stepper Motor Torque/Speed

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

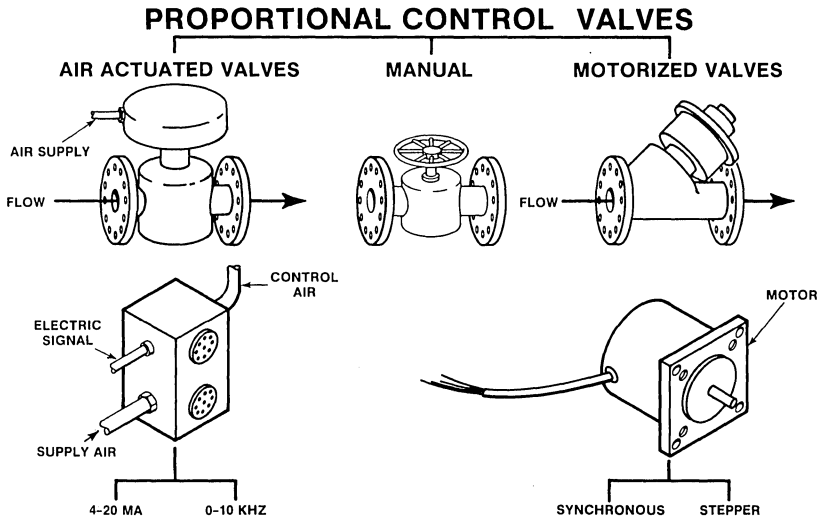


Figure 7. Control Valve Family

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

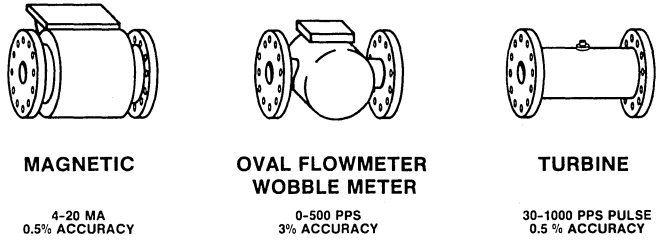


Figure 8. Flow Meter Classifications

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.

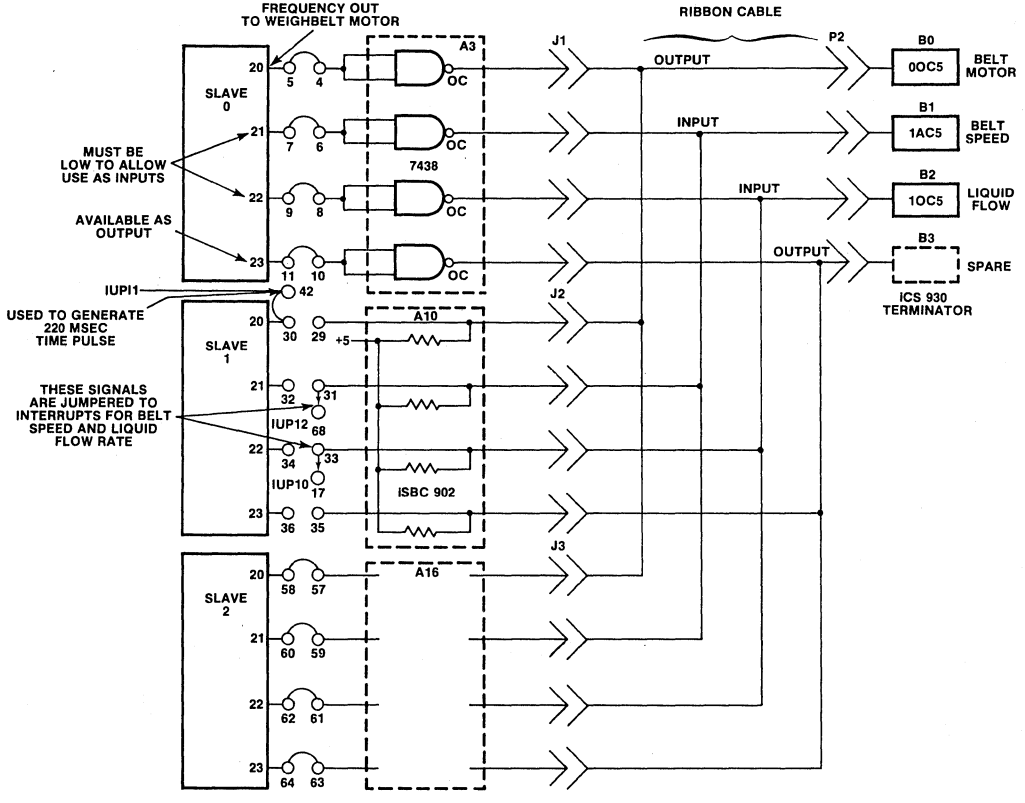


Figure 12. Port Assignments 20-23

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

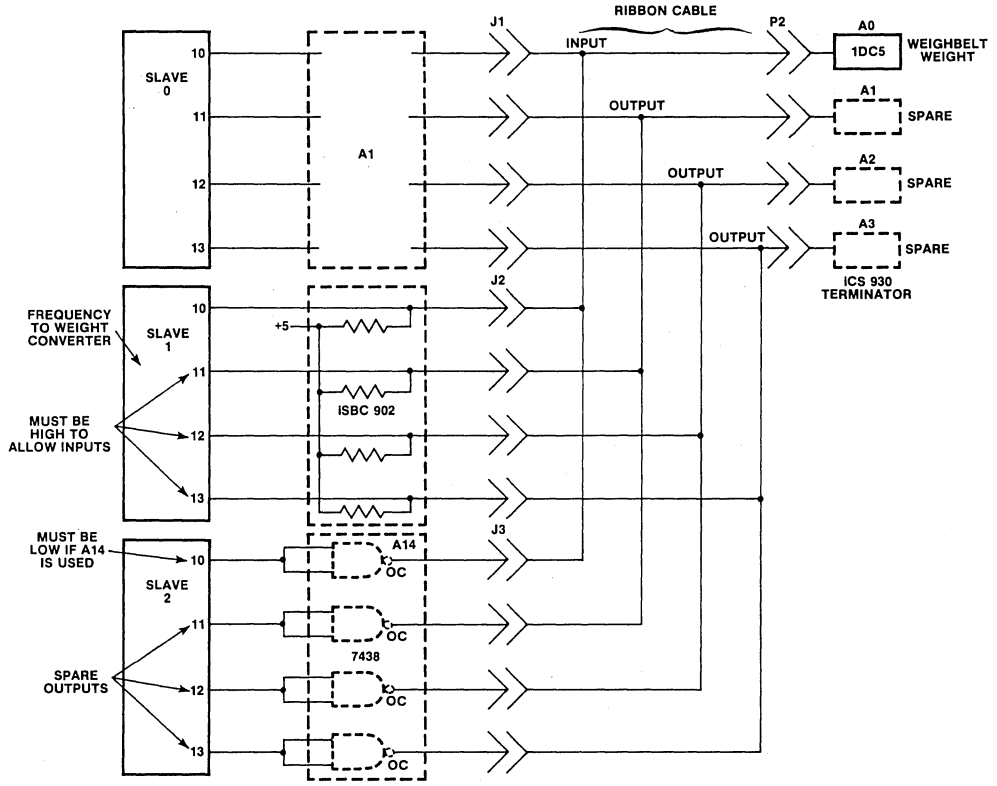


Figure 13. Port Assignments 10-13

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

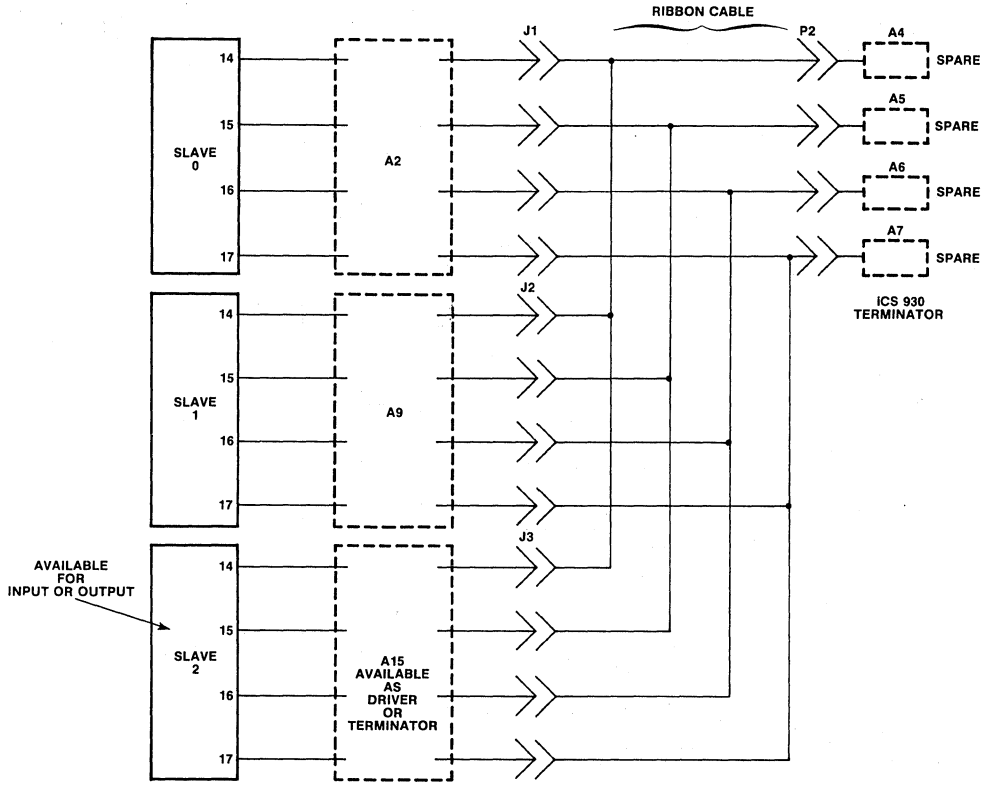


Figure 14. Port Assignments 14-17

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 minicomputer 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

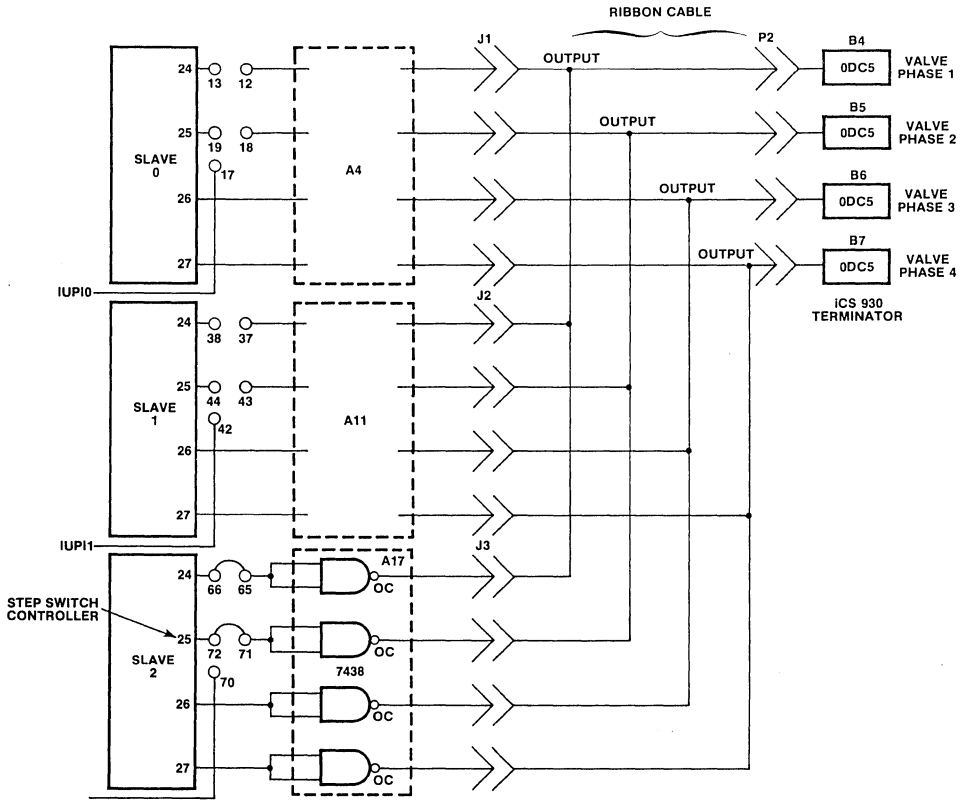


Figure 15. Port Assignments 24-27

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

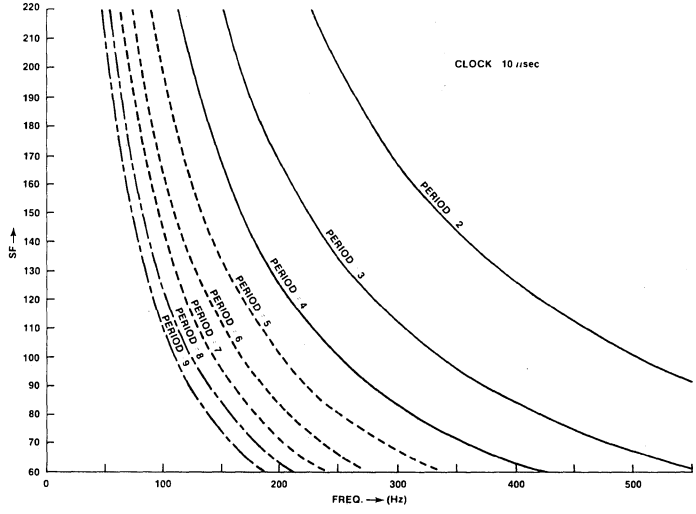


Figure 20. Frequency Vs. Parameters

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.

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

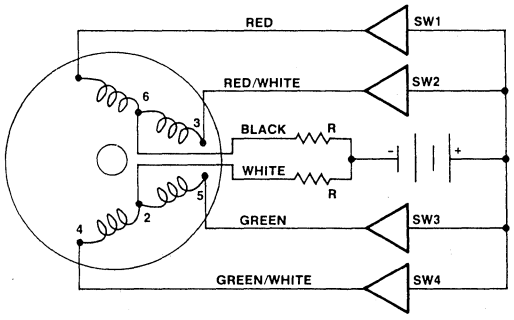
The technique used to place the processor into the desired function is the same as we have seen with the two other processors so we will not spend time dealing with the communications sequence. Instead, we will examine the techniques which can be used to determine the values of the initialization parameter 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 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 × B) where B is the number of phases



DC STEPPING CIRCUIT

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

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

Figure 26. STEPPER Motor Input Sequence

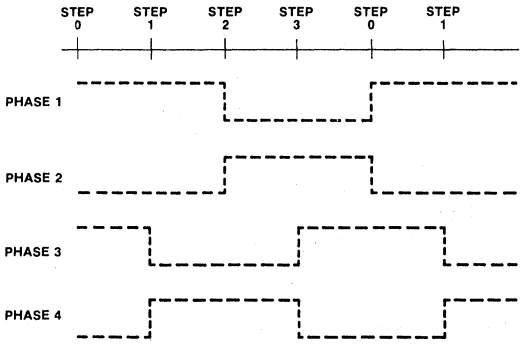


Figure 27. STEPPER Motor Waveforms

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

```
ISIS-II PL/M-80 V3.1 COMPILATION OF MODULE BACKGROUNDMODULE
OBJECT MODULE PLACED IN :F1:BCKGND.OBJ
COMPILER INVOKED BY:  PLM80 :F1:BCKGND.PLM DEBUG PAGEWIDTH(72) TITLE('BA
                      -CKGROUND PROGRAM')

                /*****************************************
                * 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 '0E5H';
    3   1           DECLARE UPI$1$STATUS          LITERALLY '0E7H';
    4   1           DECLARE UPI$2$STATUS          LITERALLY '0E9H';

    5   1           DECLARE UPI$0$COMMAND         LITERALLY '0E5H';
    6   1           DECLARE UPI$1$COMMAND         LITERALLY '0E7H';
    7   1           DECLARE UPI$2$COMMAND         LITERALLY '0E9H';

    8   1           DECLARE UPI$0$DATA            LITERALLY '0E4H';
    9   1           DECLARE UPI$1$DATA            LITERALLY '0E6H';
   10   1           DECLARE UPI$2$DATA            LITERALLY '0E8H';

   11   1           DECLARE RESET$LATCH$ADR       LITERALLY '0EAH';

                /* DECLARATION OF RAM TEST PARAMETERS */
   12   1           DECLARE BEGIN$RAM             LITERALLY '8000H';
   13   1           DECLARE END$RAM               LITERALLY '8500H';
   14   1           DECLARE ZERO$PATTERN          LITERALLY '000H';
   15   1           DECLARE ONES$PATTERN          LITERALLY '0FFH';

                /* DECLARATION OF RESET LATCH BIT ASSIGNMENTS */
   16   1           DECLARE RESET$UPI$0           LITERALLY '00000001B';
   17   1           DECLARE RESET$UPI$1           LITERALLY '00000010B';
   18   1           DECLARE RESET$UPI$2           LITERALLY '00000100B';
   19   1           DECLARE LIGHT$LED             LITERALLY '00001000B';
   20   1           DECLARE MULTI$INTR            LITERALLY '00010000B';

                /* DECLARATION OF ISBC 941 STATUS BITS */
   21   1           DECLARE IBF                   LITERALLY '00000010B';
   22   1           DECLARE OBF                   LITERALLY '00000001B';

                /* DECLARATION OF ISBC 941 COMMANDS */
   23   1           DECLARE IDEN                  LITERALLY '00H';

                /* 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 8085 PROCESSOR */
```

```
27   1          DECLARE MASKS BYTE DATA (ØØFH);

                /* DECLARATION OF PL/M-8Ø SIM INSTRUCTION */
28   1          S$MASK: PROCEDURE (MASK) EXTERNAL;
29   2              DECLARE MASK BYTE;
3Ø   2              END S$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) = Ø;

                /* 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$PATTERN;
4Ø   2              DO WHILE MEMLOC <> ONES$PATTERN;
41   3              END;
42   2          END;

                /* RELEASE 941 LOCKOUT/RESET BITS */
43   1          OUTPUT (RESET$LATCH$ADR) = RESET$UPI$Ø OR
                                           RESET$UPI$1 OR
                                           RESET$UPI$2 OR
                                           MULTI$INTR;

                /* VERIFY THAT SBC941 PROCESSOR IS IN SOCKET Ø */
44   1          DO WHILE ((INPUT (UPI$Ø$STATUS) AND IBF) <> Ø);
45   2          END;
46   1          OUTPUT (UPI$Ø$COMMAND) = IDEN;
47   1          DO WHILE ((INPUT (UPI$Ø$STATUS) AND OBF) = Ø);
48   2          END;
49   1          DO WHILE (INPUT (UPI$Ø$DATA) <> SBC941);
5Ø   2          END;

                /* VERIFY THAT SBC941 PROCESSOR IS IN SOCKET 1 */
51   1          DO WHILE ((INPUT (UPI$1$STATUS) AND IBF) <> Ø);
52   2          END;
53   1          OUTPUT (UPI$1$COMMAND) = IDEN;
54   1          DO WHILE ((INPUT (UPI$1$STATUS) AND OBF) = Ø);
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
```

```
ISIS-II PL/M-80 V3.1 COMPILATION OF MODULE MAINCONTROLMODULE
OBJECT MODULE PLACED IN :F1:CNTTSK.OBJ
COMPILER INVOKED BY:  PLM80 :F1:CNTTSK.PLM DEBUG

        $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.C0 = FEEDER$C0;
146     2              CONSTANTS1.C1 = FEEDER$C1;
147     2              CONSTANTS1.C2 = FEEDER$C2;
148     2              CONSTANTS1.C3 = FEEDER$C3;
149     2              CONSTANTS1.DT = FEEDER$TIME$INTERVAL;
150     2              CONSTANTS1.S  = FEEDER$SCALE$FACTOR;

151     2              CONSTANTS2.C0 = LIQUID$C0;
152     2              CONSTANTS2.C1 = LIQUID$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 = 0;
158     2              LIQUID$RATIO = 0;
159     2              SYSTEM$RUNNING = 0;

                    /* INITIALIZE THE CONSTANTS */
160     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 0 */
166     2              CALL PROCESSOR$0$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 */
170     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 OF              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$Ø$STATUS) AND IBF) <>Ø);
56    3             END;
57    2             OUTPUT(UPI$Ø$COMMAND)=LOOP;

                /* SET UNUSED BITS TO ALLOW EXPANSION */

58    2             DO WHILE ((INPUT(UPI$Ø$STATUS) AND IBF) <> Ø);
59    3             END;
6Ø    2             OUTPUT(UPI$Ø$COMMAND)=CLRP2;

61    2             DO WHILE ((INPUT(UPI$Ø$STATUS) AND IBF) <> Ø);
62    3             END;
63    2             OUTPUT(UPI$Ø$DATA)=INITIAL$OUTPUT;

                /* RETURN TO CALLING PROGRAM */
64    2             RETURN;

65    2         END PROCESSOR$Ø$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 'Ø33H';
67    1             DECLARE INPUT$SELECT        LITERALLY 'ØØ1H';
68    1             DECLARE OUTPUT$ENABLE$1     LITERALLY 'ØØ1H';
69    1             DECLARE SAMPLING$INTERVAL   LITERALLY 'ØØ9H';
7Ø    1             DECLARE INITIAL$STATE$1     LITERALLY 'ØE1H';

                /* 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) = Ø);
74    3                 DO WHILE ((INPUT(UPI$1$STATUS) AND IBF) <> Ø);
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 2Ø;
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 */
14Ø    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;

15Ø    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 = Ø ;

                /* 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      = Ø2Ø1H     513D
     VARIABLE AREA SIZE  = ØØØ1H       1D
     MAXIMUM STACK SIZE  = ØØØØH       ØD
     329 LINES READ
     Ø PROGRAM ERROR(S)

END OF PL/M-8Ø COMPILATION
```

```
ISIS-II PL/M-80 V3.1 COMPILATION OF MODULE PROCESSORINTERFACEMODULE
OBJECT MODULE PLACED IN :F1:OPR941.OBJ
COMPILER INVOKED BY:  PLM80 :F1:OPR941.PLM DEBUG

                $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 '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 RDFC0           LITERALLY '042H';
 15    1            DECLARE RDFC1           LITERALLY '043H';
 16    1            DECLARE PAUSE           LITERALLY '005H';
 17    1            DECLARE LOOP            LITERALLY '004H';
 18    1            DECLARE INITPF          LITERALLY '002H';
 19    1            DECLARE PACIFY          LITERALLY '001H';
 20    1            DECLARE ENFLAG          LITERALLY '006H';
 21    1            DECLARE SETOE           LITERALLY '071H';

                /* DECLARATION OF ISBC 941 STATUS BITS */
 22    1            DECLARE RFC             LITERALLY '080H';
 23    1            DECLARE IBF             LITERALLY '002H';
 24    1            DECLARE OBF             LITERALLY '001H';
 25    1            DECLARE QF              LITERALLY '010H';
 26    1            DECLARE QNE             LITERALLY '020H';

                /* 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;
 30    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 0.  TO PROVIDE MAXIMUM RESOLU-  *
               * TION, FOUR PERIODS WILL BE USED.  THE FREQUEN-*
               * CY RANGES CORRESPONDING TO EACH PERIOD ARE:   *
               *          RANGE        FREQ       RESOLUTION  *
               *            1      50 TO 165 HZ      2 HZ     *
               *            2     166 TO 225 HZ      3 HZ     *
               *            3     226 TO 285 HZ      3 HZ     *
               *            4     286 TO 550 HZ      6 HZ     *
               * THE SCALE FACTOR IS COMPUTED BY THE FORMULA:  *
               *       SF=100000/((FREQ)*(RANGE FACTOR))       *
               ************************************************/

41   1     WEIGHBELT$MOTOR$DRIVE: PROCEDURE (FREQ) PUBLIC;

           /* DECLARATION OF CONSTANT, 100,000 */
42   2          DECLARE HUNDRED$K(4) BYTE DATA (
                   0A0H,086H,001H,000H  );

           /* DECLARATION OF ISBC941 PORT ENABLES */
43   2          DECLARE ENABLE$FREQ  LITERALLY '01H';
44   2          DECLARE DISABLE$FREQ LITERALLY '00H';

           /* DECLARATION OF ISBC 941 MEMORY LOCATION COMMANDS */
45   2          DECLARE WRRM$55      LITERALLY '055H';
46   2          DECLARE WRRM$74      LITERALLY '074H';

           /* 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$0$STATUS) AND IBF) <> 0);
52   4              END;
53   3              OUTPUT(UPI$0$COMMAND) = SETOE;
```

```
54    3             DO WHILE ((INPUT(UPI$Ø$STATUS) AND IBF) <> Ø);
55    4             END;
56    3             OUTPUT(UPI$Ø$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 1ØØ,ØØØ */
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 = ØØØH;
78    4               RANGE = ØFFH;
79    4             END;

                 /* SEND NEW SCALE FACTOR TO DEVICE */
8Ø    3             DO WHILE ((INPUT(UPI$Ø$STATUS) AND IBF) <> Ø);
81    4             END;
82    3             OUTPUT(UPI$Ø$COMMAND) = WRRM$55;

83    3             DO WHILE ((INPUT(UPI$Ø$STATUS) AND IBF) <> Ø);
84    4             END;
85    3             OUTPUT(UPI$Ø$DATA) = FREQA;

                 /* SEND NEW PERIOD TO DEVICE */
86    3             DO WHILE ((INPUT(UPI$Ø$STATUS) AND IBF) <> Ø);
87    4             END;
88    3             OUTPUT(UPI$Ø$COMMAND) = WRRM$74;
```

```
 89    3               DO WHILE ((INPUT(UPI$Ø$STATUS) AND IBF) <> Ø);
 9Ø    4               END;
 91    3               OUTPUT(UPI$Ø$DATA) = RANGE;

                     /* END OF FREQUENCY OUTPUT MODE */
 92    3           END;

                     /* HANDLE FREQUENCIES < 5Ø HZ. */
 93    2           ELSE DO;

                     /* DISABLE FREQUENCY OUTPUT GENERATION */
 94    3               DO WHILE ((INPUT(UPI$Ø$STATUS) AND IBF) <> Ø);
 95    4               END;
 96    3               OUTPUT(UPI$Ø$COMMAND) = SETOE;

 97    3               DO WHILE ((INPUT(UPI$Ø$STATUS) AND IBF) <> Ø);

 98    4               END;
 99    3               OUTPUT(UPI$Ø$DATA) = DISABLE$FREQ;

                     /* END OF ALTERNATE FREQUENCY OUTPUT */
1ØØ    3           END;

                     /* RETURN TO CALLING PROGRAM */
1Ø1    2           RETURN;

1Ø2    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 Ø AND       *
                * 2ØØØ, CORRESPONDING TO A WEIGHT BETWEEN Ø.Ø AND      *
                * 1Ø.ØØ POUNDS.  EACH COUNT RECEIVED  HAS A VALUE      *
                * OF Ø.ØØ5 POUNDS.                                     *
                *******************************************************/

1Ø3    1       WEIGHBELT$WEIGHT: PROCEDURE ADDRESS PUBLIC;

                /* DECLARATIONS OF VARIABLES USED IN THE PROCEDURE */
1Ø4    2           DECLARE (LCOUNT,HCOUNT) BYTE;
1Ø5    2           DECLARE WEIGHT ADDRESS;

                /* GET INPUT COUNT LOW BYTE */
1Ø6    2           DO WHILE ((INPUT(UPI$1$STATUS) AND IBF) <> Ø);
1Ø7    3           END;
1Ø8    2           OUTPUT(UPI$1$COMMAND) = RDFCØ;

1Ø9    2           DO WHILE ((INPUT(UPI$1$STATUS) AND OBF) = Ø);
11Ø    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$WEIGH;
                $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 1ØØ 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 'ØØ5H';
138    2            DECLARE REVERSE   LITERALLY 'Ø8ØH';

                /* IF NO MOVEMENT, SKIP OPERATIONS */
139    2            IF POSITION <> Ø
                    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) <> Ø);
147    4                END;

                /* REQUEST DESIRED STEP RATE */
148    3                DO WHILE ((INPUT(UPI$2$STATUS) AND IBF) <> Ø);
149    4                END;
150    3                OUTPUT(UPI$2$DATA) = STEP$RATE;

                /* REQUEST STEPPER MOVEMENT */
151    3                DO WHILE ((INPUT(UPI$2$STATUS) AND IBF) <> Ø);
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
```