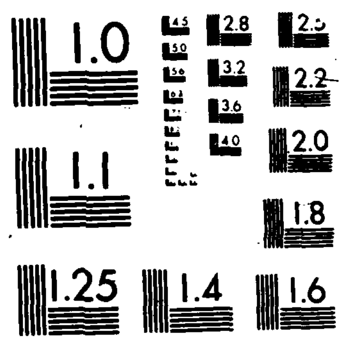
AD-A172 023 DESIGN AND PARTIAL IMPLEMENTATION OF A COMPUTER 1/5
CONTROLLED DATA COLLECTION SYSTEM(U) AIR FORCE INST OF
TECH WRIGHT-PATTERSON AFB OH SCHOOL OF ENGI.. L E LUTZ
UNCLASSIFIED FEB 86 AFIT/GE/ENG/86M-1 F/G 9/2 NL


1.0
2.8
2.2
3.2
3.6
4.0
2.0
1.1
1.8
1.25
1.4
1.6

DESIGN AND PARTIAL IMPLEMENTATION OF A COMPUTER CONTROLLED DATA COLLECTION SYSTEM

THESIS

Lloyd E. Lutz Jr.
Captain, USAF

AFIT/GE/ENG/86M-1

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GE/ENG/86M-1

DESIGN AND PARTIAL IMPLEMENTATION OF A COMPUTER CONTROLLED DATA COLLECTION SYSTEM

THESIS

Lloyd E. Lutz Jr.
Captain, USAF

AFIT/GE/ENG/86M-1

AFIT/GE/ENG/86M-1

DESIGN & PARTIAL IMPLEMENTATION OF A COMPUTER CONTROLLED DATA COLLECTION SYSTEM

THESIS

Presented to the Faculty of the School of Engineering
of the Air Force Institute of Technology
Air University
in Partial Fulfillment of the
Requirements for the Degree of
Master of Science in Electrical Engineering

by
Lloyd E. Lutz Jr.
Captain USAF
Graduate Electrical Engineering
February 1986

Preface

This thesis describes the design and software for a computer controlled data collection system. A MCB Z-8Ø development system with an AIO analog input board was both the target hardware for the data collection system and the computer system the software was developed on. The software described in this thesis is a mixture of PLZ, a Pascal like language, and Z-8Ø assembly language with hooks from both into the development system's RIO Operating System.

The software doesn't implement the full design and isn't completely bug free. The difficulties of too little time and too much code to debug took their toll. I have flagged weak points and logged my suspicions where appropriate in the code descriptions.

Special thanks go to the Apple Computer Company for their development of the Macintosh, LaserWriter, and MacWrite. Without these products this thesis would never have been written.

Lloyd E Lutz Jr

Contents

List of Figures

Figure	Figure Name	Page

Figure	Figure Name	Page

Abstract

A computer controlled data collection system was designed and partially implemented in software. The design concept is for a data collection unit to be placed inside the system being tested where it stores the test data in an internal memory. Post-test this internal unit is connected to and polled by an external control and data storage unit which archives the data. Both units are computers. This combination of an internal data collection unit and an external control and storage unit is intended for testing applications where it is either undesireable or not possible to connect the sytem being tested to external data recording devices during the test event.

The partial implementation of this dual unit data collection system design was performed on a Zilog MCZ Z-8Ø development system in PLZ, a Pascal-like language, and Z-8Ø assembly language. Routines to improve the input / output and hardware access of PLZ were written and used. The software to implement the internal data collection unit and portions of the external control and data storage unit were also written. The internal unit routines employ a Zilog Counter Timer Circuit to generate sampling period interrupts. The analog to digital conversion is accomplished via a Zilog Analog Input Output (AIOI) board. The data collection system is not fully operational.

I. Introduction

Whenever a system is tested, a major part of the activity is collection of performance data. "Did it work ?" is not a question answered by the outcome of test alone. Rather it is answered by an evaluation based on the information collected during the test. In the past, this performance data might have been manually collected, notes carefully recorded in an lab book, or as a photographic image of an oscilloscope trace. Today's technology permits the collection and storage of performance data in electronic forms, both analog and digital. Besides automating the data collection process, this electronic collection of data permits analysis without having to manually reenter the data into computers.

The automated collection of performance data is accomplished by attaching sensors to the system under test and then connecting the sensors to some data recording equipment. The sensors translate the physical responses of the system being tested into electrical signals. Examples of sensors include strain gages for movement and pressure; current, electric field and magnetic field sensors for Electromagnetic Pulse testing, and microphones for human speech. The data collection equipment stores the sensor generated signals. Examples of recording equipment include tape recorders and transient digitizers like Tektronix 7912s. The connection between the sensor and the data recording equipment can range from simple twisted pair wiring to multiplexed fiber optic links to the RF data links from tagged grizzly bears through the TDRSS satellite to NASA's ground stations. The length and type of connection used is dependent upon the nature of the system being tested.

There are instances in testing however where it is either physically impossible or undesirable to connect the item under test with some external data recording system. For example, the "black boxes" of airplanes, the cockpit voice recorders and the flight data recorders, are internal to the system. It is not feasible to hard wire aircraft to ground based recorders or squander the RF spectrum on data links. Another example is Electromagnetic Pulse (EMP) testing. External data recorders can not be wired to sensors in the aircraft undergoing EMP testing for the presence of these conductors alters the EMP response of the aircraft. Use of dielectric instrumentation cables, like fiber optics is one solution, though this tethers the test object. RF links are also possible though complex to set up and often limited in bandwidth. Another solution exists and is used. The sensor data is stored within the system being tested and then extracted after the test event is over. In the first example, the flight recorders are recovered from the crashed aircraft; the crash being the test event. In the EMP example, an early procedure was to put oscilloscopes with cameras inside shield boxes (EMP & noise "proof" enclosures) and place these boxes within the aircraft; the exposed film was recovered after each EMP exposure (Ref 12). In both cases, the data is saved in recording equipment placed inside the system being tested and then the data is retrieved after the test is over.

This thesis investigation considers another version of the internal data storage approach discussed above. The sensors on the item under test are connected to a data recording unit located inside the test item as shown in Figure 1 below. This internal data collection unit is a microprocessor / memory system that samples sensor data at a programmable rates and saves the data in random access memory (RAM). Pretest, the internal data collection unit is programmed for the desired sampling by the external control and data storage unit. Next, during the test, the links between the internal unit and the external unit are severed or ignored. Post-test, the internal data collection unit is reconnected to the external control and data storage unit. The data is then transferred out of RAM to the external unit and saved in some long term storage medium like a floppy disk or data tape. The external control and data storage unit would also handle simple data scaling and printing of the data and could be available for User data manipulations as well.

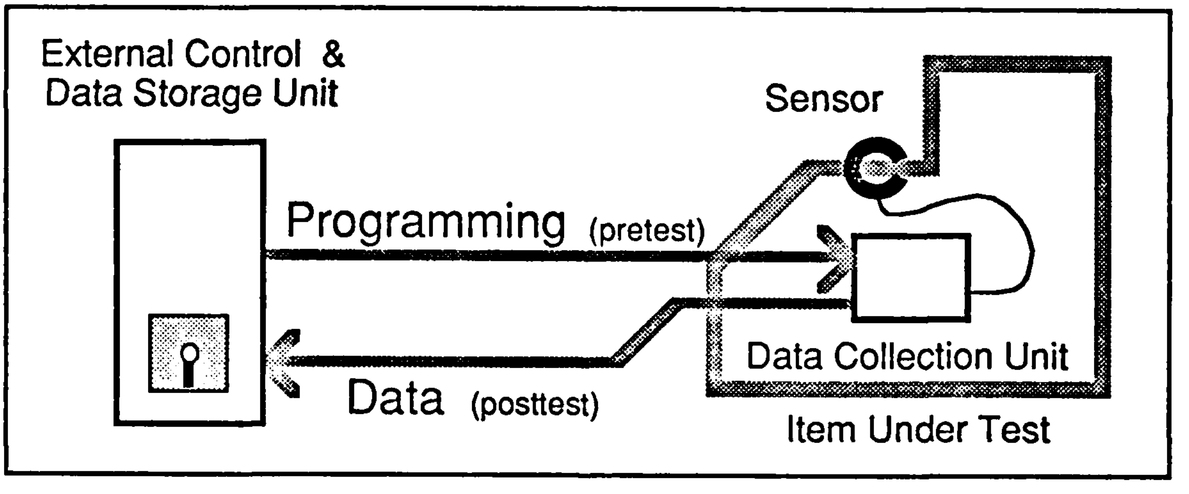


Figure 1. Data Collection System

Both the internal data collection unit and the external data storage system are digital devices , adapted through their software for the specific needs of each data collection effort. The object code of the collection unit, would likely be ROM based; for the storage system the object code would probably be called from disk. The key is that the collection unit and the storage system must communicate with each other based on a common understanding of purpose. An example of this type of system is the Tektronix 7912 and a post test polling computer (Ref 11). This thesis deals with the software required to make such a system work (in conjunction with the hardware of the system), the software of the internal data collection unit and the external data storage system.

Requirements of Data Collection System

While it is simple to state the purpose of a data collection system, "To Collect Data", it is more important to examine the characteristics or attributes required of the system. The primary attributes of concern for this data collection system are accuracy, data integrity, flexibility, and a simple user interface. In practice, it is vital to quantify the specific requirements for each attribute; to define exactly what the necessary performance characteristics are. As this thesis is not tageted to any specific application, the following discussions of accuracy, data integrity, flexibility, and a simple user interface are general.

Accuracy

For a data collection system to have any value, the data it collects must as accurately as possible represent the original physical phenomena or sensor signals that were sampled. There are three facets to this requirement for accuracy: amplitude fidelity, sampling period, and data scaling. The need for amplitude fidelity will be discussed first.

The mapping between the amplitude of the analog signal being sampled to the digital values stored has several variables: analog to digital (A/D) conversion fidelity, linearity, and sensor impacts. First, the analog to digital (A/D) conversion must have as much fidelity as possible. Obviously more bits do yield greater fidelity. This increased fidelity is paid for in increased hardware costs or settling times. The objective in selecting the number of bits of the A/D converter is to match its conversion range to the data signals of interest. This matching has two aspects, maximum amplitude (or dynamic range) of the input signal and resolution (units per least significant bit or scaling) required by the test application. Any A/D converter can be matched to the maximum expected amplitude of any given signal through the use of attenuators or amplifiers; the resolution of amplitude is another matter however. Figure 2, below, shows the resolution variation for a variety of A/D converter sizes and signal dynamic ranges. The selection of the size of the converter must be based on the expected dynamic range of the signal and the resolution or scaling required. If there is a mismatch in dynamic range , the analog signal may overflow the A/D converter or the signal may register only in the least significant digits. Through correct matching of the input signal to a properly sized A/D converter with amplifiers or attenuators, the amplitude of the analog signal will be accurately represented with the resolution necessary for the specific signal or sensor of interest.

Resolution of Least Significant Bit

Signal Range \ Total Elements	Size of Analog to Digital Converter 4-Bit	8-Bit	12-Bit	16-Bit
	16	256	4,Ø96	65,536
Ø to 1	.Ø63	.ØØ4	.ØØØ2	.ØØØØ2
Ø to 1Ø	.625	.Ø39	.ØØ24	.ØØØ15
Ø to 1ØØ	6.25	.391	.Ø244	.ØØ153
Ø to 1,ØØØ	62.5	3.91	.2442	.Ø1526

Figure 2. Resolution of Least Significant Bit for Various Sized Analog to Digital Converters and Input Signal Ranges

A second variable of the mapping between the analog signal's amplitude and the digital values is linearity. It is desirable for the bit change to be the same for a signal amplitude change regardless of where in the dynamic range the signal change occurs. The delta bits for a 98.7 to 99.Ø volt change should be the same as for a 7.3 to 7.Ø volt change. This linearity is a function of the A/D converter and any signal conditioning equipment (attenuators or amplifiers). Two courses of action are available, get as linear a system as possible or measure the nonlinearity and extract its effects posts test.

The third variable is the sensor itself. Though sensor concerns are outside the scope of this thesis, the resolution of the sensor must be matched to the physical phenomena being measured. As with the A/D converter, dynamic range and resolution are of concern. Linearity is a concern for the sensor also.

The second facet of the requirement for accuracy centers on the sampling period. Of foremost concern is that the sampling rate be sufficient to capture the frequencies of interest in the input analog signal. Beyond the sampling rate, are two aspects of concern for the generation of the sampling period. First, the sampling period employed must be stable; that is the time between samples is constant from the beginning of data collection through the end. This is dependent upon the clock used to trigger individual samples. The second aspect of concern is that the sampling period employed should be what was specified. Few things could distort test findings more than to have the time base unknowingly off. Correct implementation of the specified sampling period is a function of both the clock used and the routine that translates the specified sampling period into hardware controls or programming.

The preceding paragraph discusses continuous samplers. Another type of sampler, event driven, also exist. Event driven samplers collect data only when something of interest occurs. For this type of sampler, timing accuracy centers on knowing when the event occurred. Event driven samplers are not within the scope of this thesis effort. However, for continuous samplers, knowledge of when the sampling began, or a time tie between the sampling interval and some external event is valuable.

The third facet of accuracy requirements is data scaling. Each step from the actual physical phenomena to the digital data stored alters the representation of the phenomena. A pressure of 3 KPa is translated by a sensor into a 3 volt signal; an amplifier boosts this to 27.3 volts; a 12 bit A/D converter transforms it into the binary string 1ØØØ1Ø1111ØØ. Data scaling is the process of converting this binary string back into physical parameters. The process can be as straight forward as multiplying the digital value by a single scale factor. It could be a complicated filtering effort involving multiplication by an amplitude dependent scale factor to remove nonlinearities produced by the sensor. In either case, the scaling process must be uniquely accomplished for each sensor to satisfy this third facet of accuracy.

Data Integrity

Data integrity is the second attribute required of a data collection system. Data integrity simply refers to the data being protected from loss or alteration from improper or inadvertant actions. Tests are not inexpensive and to loose test data or have it altered could force a retest or perhaps acceptance of the loss of unreproducable data (aircraft flight data recorders for example). Also including in data integrity is data traceability. As files of data are manipulated, it is vital to know what the original raw data file was and which file was the immediate parent of the manipulated file.

Flexibility

A data collection system having the attribute of flexibility is a system that can readily adapt to changing data collection efforts. This attribute can be examined from two perspectives, the flexibility to adapt to different systems being tested and the flexibility to adapt to changing needs during the test of a single system.

It makes sense for a piece of test equipment , such as a data collection system, not to be tailored to a single specific system under test. If it were tailored it would have to be developed anew for each new system being tested. Instead, the data collection system should be sufficiently broad in its capabilities to support a reasonably wide range of applications. This could mean being able to withstand both the g-forces of an aircraft and the thermal environment of a tank in desert testing. This could mean being able to record information from both a current transformer hooked to a high voltage line and a strain gage on a tactical shelter during an overpressure test. This could mean being able to both record 1,000 samples in 10 seconds of transient response measurement or one sample every 10 seconds of long term stability measurements. In the actual development of a data collection system, the scope of application would have to be clearly defined in order to establish firm design requirements. The following are examples of the kinds of variations a general purpose data collection system (intended to be located inside the object under test) would have to accomodate.

Varied Test Environments. The data collection system, particularly the internal unit, should be able to operate in many environment such as high and low temperature, electrical noise, RF, salt water atmosphere, pressure, dynamic loads, high humidity or wet environments, and shock or vibration. Producing hardware that can function in these environments is mostly a construction and packaging problem.

Assorted Sensors. A unit intended for multiple purposes must be able to interface with many different kinds of sensors. The inputs may be differential or single sided. The sensor output voltages may be in a millivolts range or 10's of volts. The impedances of the sensor and the collection unit must be matched.

Range of Sampling Periods. Sampiing periods range from well above 10^6 samples per second for nuclear weapon effects (Ref 11) to less than one sample per hour for thermal drift. The higher sampling rates will force the use of faster analog to digital converters, processors, and memory.

Number of Samples. The number of samples needed will vary greatly based on two factors, the sampling period needed and the frequency with which the stored data can be extracted. A system requiring only one sample per hour could go nearly a whole year on 8K bytes of memory. On the other hand, at $1Ø^6$ samples per second, 64K of memory would be filled in less than 1ØØ milliseconds. The need for large numbers of samples will rapidly complicate the internal data collection unit. More memory means greater power consumption, more complicated addressing, and larger physical size. The longer the interval between extraction of stored data, the larger the memory needs to be.

Frequency of Access. This refers electrical access for retrieval of data or charging of batteries. If the access is infrequent, then the power supply will have to sufficiently large to power the unit between accesses. The frequency of access also impacts the size of the data storage memory as discussed above.

Physical Dimensions. As the data collection unit will be located within test objects, It should be as small as possible and be readily mountable.

In the actual design and implementation of an internal data collection unit, trade-offs would have to be made between the above capabilities and other parameters such as cost. For example, one test environment, nuclear radiation testing, imposes significant problems for electronics. To include this environment upon a general purpose data collector would impose severe penalties on other applications. Extremely long sampling periods or very large numbers of samples are other examples of needs outside the bounds of a "general purpose" data collection system. For these kinds of requirements, specialized units would probably be created; these special needs just push hardware flexibility too far. The software however, would be consistent.

The second perspective on the required attribute of flexibility is the ability of the data collection system to adapt to changing needs during the test of a single system. Perhaps predictions were off and signals that were expected to be 1Ø's of volts are actually just a few volts. To have the capability to remotely adjust the internally mounted data collection unit is quite desirable. With remote adjustment or programming the test apparatus wouldn't have to be torn apart to make adjustments. Perhaps the item under test itself is inaccessible except via control lines. It is desirable to have the capability to change the following items remotely.

Input Channel/Sensor Selection. With the ability to remotely shift between channels or sensor, a single internal collection unit could perform the function of several. This is an advantage only for reproducible tests.

Attenuator / Amplifier Selection. The ability to change the gain remotely is vital. The example above of errored predictions shows an application where remote adjustment of attenuators would be useful.

Sampling Rate. As a unit is switched between sensors or to accommodate different test interests, the sampling period of the unit needs to be changed. For example, one test run might be made at 1K samples per second to measure the initial transient response followed by a second test with 10 samples per second to examine the long term response.

Number of Samples. Given the variation in sensors and sampling rates, the number of samples collected needs to be remotely controlled.

Mode of Operation. Should the test go into a hold, it would be useful to place the internal data collection units into some standby state to conserve battery charge. Other states of interest would be battery charge, programming, off, and ready, self-test/readiness check.

Flexibility is thus a two perspective attribute of the requirements for the data collection system. The ability to adapt to both varied test requirements and changes in an ongoing test are needed.

Simple User Interface

The final requirements attribute of the data collection system is that it have a user interface. This involves four factors, clear instructions, error diagnostics, operation on the users' terms, and fool tolerance. The first two factors revolve around the messages passed to the user. Instructions must clearly spell out what the user is to do and the format it should be done in. Error messages must tell what went wrong, where it went wrong, and, if possible, why it went wrong.

Operation on the users' terms refers to two efforts. First, all commands need to be in "real world" terms, not values selected for ease of programming. For example sampling periods should be specified in seconds, not clock cycles. By requesting and expressing information in terms readily understood by the users, the ease of use of the system is greatly enhanced. The second portion of operation on the users' terms is for the computer to do the work. If translations between units are required, the routine should perform the translation rather than forcing the users to do so. Also included is telling the users what is happening. Nothing disturbs a person more than sitting by a computer which hasn't "said" a thing for several minutes. Status feed back is important.

The final factor of a simple user interface is fool tolerance. While no system can be made totally fool proof, reasonable steps can be taken to avoid problems. The factors already discussed go a long way towards fool tolerance; the remaining step is error checking on the user input. Are the users' input commands in range, consistant, of the proper format, and complete? If not tell the users what is wrong and remind them of the allowable inputs. These steps will not fool proof the system (the reset switch will still get bumped) but they will greatly reduce the occurrence of inadvertant errors.

In summary, the data collection system must be accurate, must ensure data integrity, must have sufficient flexibility, and must present a simple interface to the users of the system. Though there are design trade-offs within and among these requirements, a reasonable general purpose data collection system design can be derived from them.

Hardware Used for this Thesis Effort

Were this thesis effort an actual development of a data collection system, a substantial portion of the effort would center on the selection or design of the hardware which implements the collection system. For this thesis effort, the hardware was a given. The thesis effort focused on the design and implementation of the software needed to make the data collection system functional.

The hardware used for this thesis effort was a Zilog MCZ-8Ø development system. It was used for several reasons. It was available, it had an analog to digital (A/D) converter board, timing chips for generating sampling intervals were present, a high level language similar to Pascal was available, and extensive assembly language programming tools were available. Thus the MCZ system met many of the requirements for the data collection system discussed in the previous section and provided the software development tools needed to carry out the thesis effort. The following is an overview of the MCZ system used as both the software development system and as the target hardware for the data collection system (Refs 1 through 9).

The MCZ development system consisted of

1. Equipment chassis with power supply and card cage
2. Two 8 inch floppy disk drives.
3. A Zilog MCB Microcomputer board. This board held:
   - Z-8Ø microprocessor
   - 3K ROM with monitor routine
   - System Clock
   - 16K of RAM
   - Z-8Ø CTC (Counter Timer Circuit)
   - Z-8Ø PIO (Parallel Input Output)
   - USART (Universal Synchronous Asynchronous Receiver Transmitter)
4. A Zilog MCD Board (memory & disk controller) 48K RAM
5. A Zilog SIB Board (serial interface board, has three CTCs)
6. A Zilog AIO Board (analog input output card) which has a 12 bit analog to digital converters.
7. RIO Operating System which includes disk operating system.
8. An ADAM-3 terminal.
9. A NEC Spinwriter Printer.

To prepare the MCZ system for this thesis effort, the AIO board had to be integrated into the system (Ref 8:Sec 2, Sec 3:5); it had never been installed. Installing the AIO board required minor rewiring of the motherboard of the card cage, addition of backplane connecters for the AIO board's interfaces, and the fabrication of connection board to permit easy hookup to the AIO board's interfaces. Once the AIO board was installed, its disk based diagnostics were run and the board's alignment was checked and adjusted as required (Ref 8: Sec 5).

As target hardware for the data colection system, the MCZ hardware met several of the data system requirements discussed in the previous section. The system possessed accuracy with both a 12 bit analog to digital converter and ample hardware for generating accurate sampling periods (Ref 2, 7, and 8). The RIO operating system supported disk file operations permitting protection of data integrity (Ref 4). The system was relative flexible having sixteen input channels for the analog to digital converter (Ref 8:1) and sufficient memory for both the programs and data sample storage. The Adam-3 terminal would serve as the user interface; the bulk of the simple user interface up to the software.

While the MCZ system met many of the requirements for the data collection system, it did not mesh well with the hardware concept of the data collection system. The MCZ system is a single system; the data collection system concept calls for two distinct hardware units, the internal data collection / temporary storage unit and the external control / archival storage unit. This mismatch between the realities of the MCZ hardware and the hardware concept of the data collection system is largely resolved in software.

The focus of this thesis effort is the software required to make the data collection system work. Thus the reality that a single set of hardware was being used could be masked, in part, by making the software of the two data collection units separate and distinct. As will be shown, the software developed for this thesis effort maintains this division. The software of the internal unit does not talk directly to the user except for a trigger signal. The software of the external unit does not have direct access to the analog to digital conversion. The programming the external unit provides to the internal unit is represented by the parameters passed between the software of the external unit and the software of the internal unit. Thus, while a single set of hardware is used, the software of the data collection system is separated into internal and external units.

Overview of System Design

In designing the software of the data collection system software, the first question was "What tasks will the user need to accomplish ?" The principal task is to collect the data, but what else would the user need to do. Three specific tasks and one general activity area were fidentified. The data read in from the internal unit and stored in the external unit is raw data. It is in the digital form received from the A/D converter. Thus, an important task is to translate this raw data into data set in real world terms; the raw data needs to be scaled. To maintain data integrity, this scaled data should be written into a new file leaving the original raw data file unchanged. To accomplish the scaling, the user must specify the scale factor to be used; a unique scale factor for each input channel. Thus a third type of file is needed, a file of scale factors. Having translated the information of the raw data file into the information of the scaled data file, the users would probably want to print out the data or perform further manipulations of their design or choice. The printing out of data is the third specific task of the data collection system and the user defined manipulations are the general activity area. One final feature of the data collection system is a common user interface so all the the tasks can be invoked in a consistent fashion. Thus the five tasks the data collection system must accomplish are

Collection and Storage of Data
Setup of Scale Factors in a File
Produce a File of Scaled Data from the Raw Data and Scale Factors
Output of Data Files (Both Raw and Scaled)
Support User Manipulations of Data

all with a consistent user interface.

Figure 3A below shows how these five tasks or processes interact with each other, the operating system, and the user. The common user interface is present as an interface or interpreter between the operating system and the processes of the data collection system. Though the figure implies that all operating system calls would go through the user interface, this is not necessarly the case. Once a process begins its execution, standard operating system calls and messages to the user would go directly to the operating system rather than through the interface. Thus elements of the common user interface are implemented through out the processes of the data collection system.

Figure 3B shows the elements used to implement the data collection system shown in Figure 3A and shows the hierarchy of these elements. The process of Collect and Store Data of Figure 3A is implemented by Collect_Data Module and Sampler Module along with the hardware elements and calls to the operating system. The Set Up Scale Factor Filed process is implemented by the Scale_Factor Module of Figure 3B, again assisted by operating system calls. The Figure 3A processes of Scale Data, Output Data, and User Data Manipulations, were not implemented. Shown in Figure 3B as portions of the operating system are three modules of general support software which were implemented as software development aids. The common user interface of Figure 3A is partially implemented and is represented in Figure 3B by some of the calls to the operating system from Collect_Data Module and Scale_Factor Module.

In the following paragraphs, the activities performed by each process of the data collection system and, when appropriate, an overview of how the process was implemented the will be presented. The purpose and design of the general support software will also be presented. Information of greater detail on design and implementation for each module is presented in later sections of this thesis for each software module. Please note that the software developed for this thesis effort addresses only the Data Collection and Storage process and the Set Up Scale Factor File process. These processes were implemented since their output is required as input to the remaining processes. Also, the remaining processes are simpler to implement and can be, in part, built from the routines of the implemented processes.

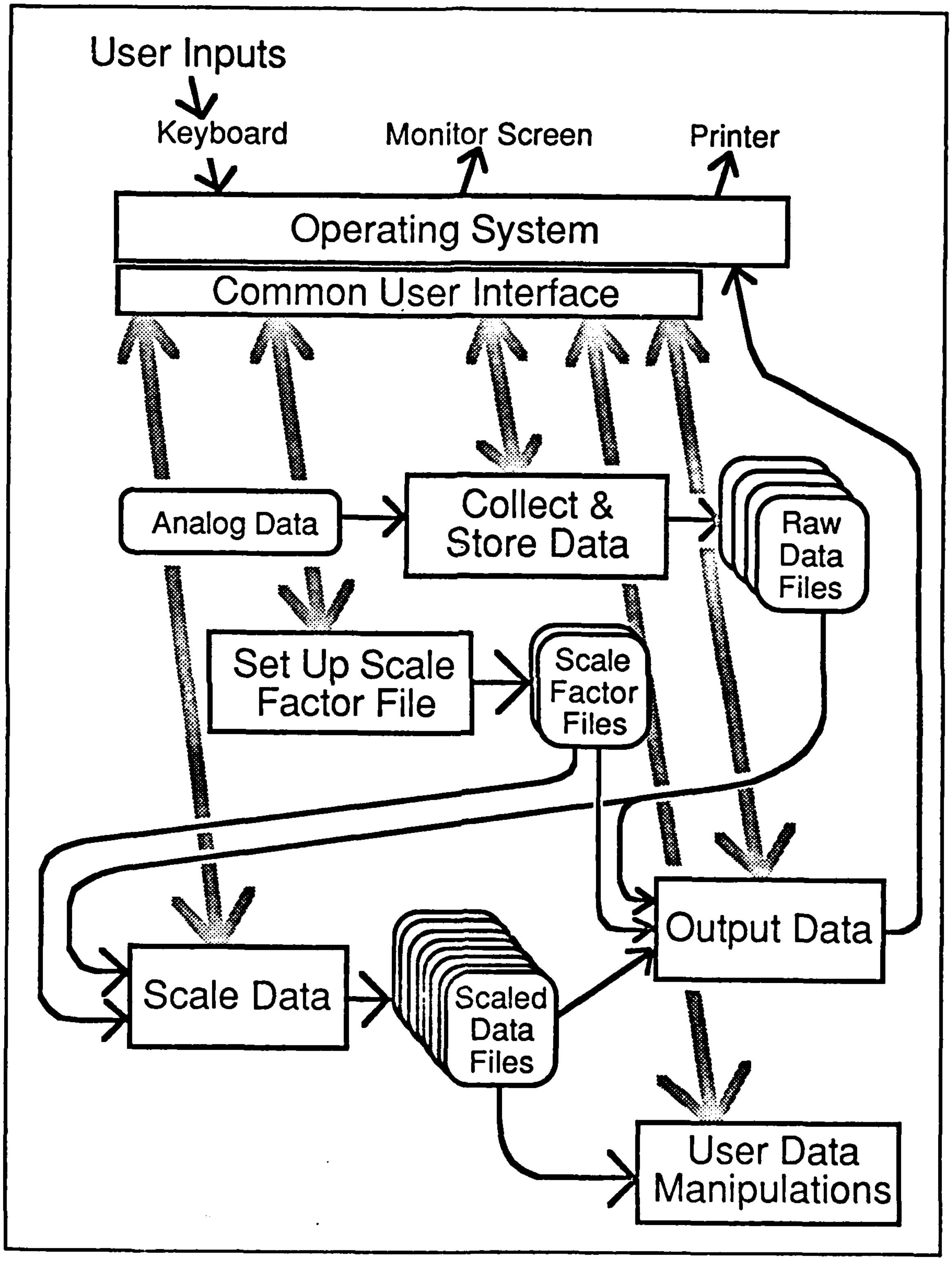


Figure 3A. Data Flows Between the Major Processes of the Data Collection System, the Operating System, and the User.

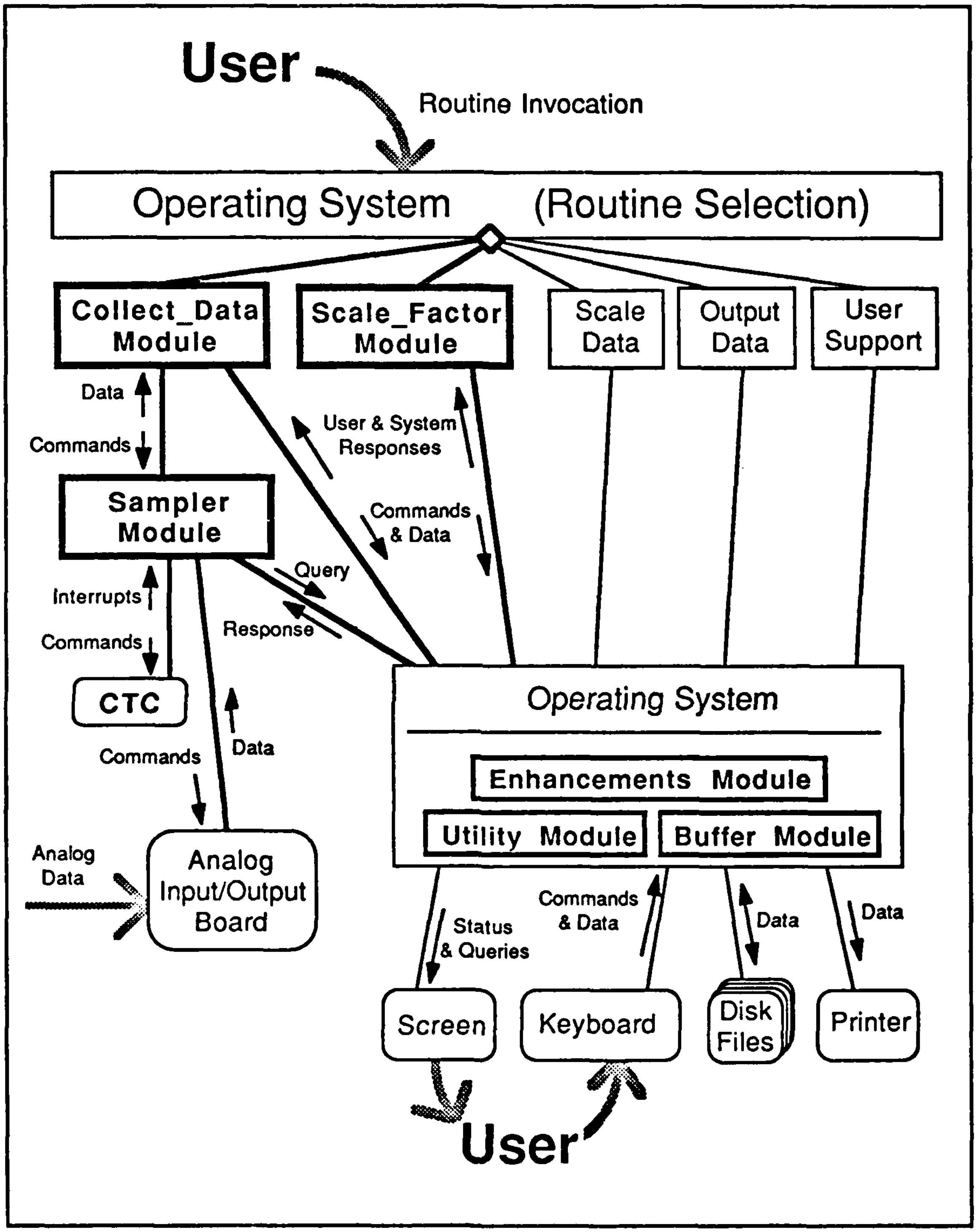


Figure 3B. Hierarchical Relationships Between Components of the Data Collection System Components, the Operating System, and the User.

General Support Software

Early in the design process one of the questions raised was which language should this data collection system be implemented in, PLZ, Zilog's Pascal like language, or Z-8Ø assembly language. PLZ offered some of the benefits of a high level language such as mathematical operations. PLZ however was quite cumbersome in the string input and output which would be required for the user interface. Assembly language would be fast and offered direct access to input output ports,memory, and the Z-8Ø registers. On the other hand, Z-8Ø assembly language was unfamiliar, IO was even harder than PLZ, and math operations would be far more difficult. The selected approach was to use the best qualities of both PLZ and Z-8Ø assembly language plus providing some software "improvements" to PLZ. The software improvements focused on two areas, string input and output, and on access to system hardware. The string input and output improvements became the Enhancements Module; the hardware access routines became the Utility Module.

The PLZ Language routines of the Enhancements Module were written to make string input and output easier in PLZ. The routines were written to approximate the standard Pascal read and write statements (Ref 10: Sec 7.2). The major difference is that the Enhancements Module routines all have in input parameter for the logical unit number, where Pascal handles device specification as an optional parameter with the compiler sorting things out. The PLZ compiler was not capable of this. The choice for the Enhancements Module was to use a mandatory logical unit parameter or add new routines and global variables to switch between logical units. The logical unit parameter approach was selected as it is closer to the Pascal implementation and would yield far more readable code. The Enhancements Module routines were fully developed and tested.

The Utility Module assembly language routines were initially written to give PLZ language programs access to the AIO board. This purpose was expanded to give PLZ language routines access to other portions of the system not normally accessible to PLZ. The module ultimately contained nine assembly language routines. They provide access to input/output ports, individual memory locations, the system date, the operating system memory manager, and the enabling / disabling of the Z-8Ø CPU interrupts. The nine assembly language routines of the Utility Module were completely developed and tested.

With the "improvements" provided by the Enhancements Module and the Utility Module, development of the data collection system software could begin.

Collect and Store Data

This process is the heart of the data collection system for it is in this process that the analog data is collected, converted to digital data, placed in temporary storage, transferred to the external data storage unit, and archived on magnetic media. The design of this process and its implementation in software focussed on two competing sets of constraints. First, the design looked the requirements for the data collection system discussed in the previous section. Second, the design had to live within the constraints of the Zilog MCZ development system. In addition, the software of the internal data collection / temporary storage unit and the software of the external control / archival storage unit had to be separate and distinct to keep faith with the hardware concept of the data collection system. The design process for the Collect and Store data process looked at three basic areas, the analog to digital conversion, the timing of the sampling periods, and the archival storage of the converted data.

Analog to Digital Conversion. The design of the analog to digital conversion portion of the Collect and Store Data process was based on the capabilities of the AIO (Analog Input Output board). This board satisfies many of the requirements outlined in the opening section. The board has a 12 bit analog to digital (A/D) converter; this meets the needs for accuracy. Via programming, the board can address any one of sixteen input channels; this meets the flexibility need for in place adaptability. The A/D converter settles in about 2Ø microseconds (Ref 8: Sec 3.5.5). Giving a liberal allowance for program overhead this permits a minimum sampling period of about 5Ø microseconds, a reasonable minimum for a general purpose data collection system. The board is hardwired for +/- 10 volt full scale inputs and coding the output in two's complement format (Ref 8: Sec 3.5.1).

Given the capability of the AIO board, the method of employment was determined. The AIO board would be programmed into a polled input mode (Ref 8: Sec 4). Then, upon receipt of a timing signal, the desired input channel number would be written to the board; this initiates an A/D conversion. The controlling program then goes into a loop, polling the AIO board's status register until the data ready flag is raised. The controlling routine then reads the data from the AIO board and stores it in memory. This sequence is repeated for each timing pulse. Initial design of the software was accomplished in PLZ. This initial software is the AIO.PLZ.S Module. For the final program, assembly language was selected for reduced overhead and simpler handling of the timing pulses. The assembly language program which, among other things, implements this process is the Sampler Module, the software of the internal unit of the data colection system.

Timing of Sampling Periods. The second general area of the Collect and Store Data process is the selection and generation of the sampling periods. The implementation of this timing is based on the timing capabilities of the CTCs (Counter Timer Circuit) of the MCZ Development System's SIB. The CTC can be easily programmed to generate periodic interrupts with intervals of 6.515 microseconds to 26.68 milliseconds (Ref 7: Sec 3.7). This timing capability meets the needs of accuracy and begins to satisfy the requirement for flexibility discussed in the opening section. The 26.68 millisecond maximum however is not sufficiently long for a general purpose data collection system. So, a sixteen bit counter was added. The combination of the CTC timer and a sixteen bit counter yields a maximum timing period of 29.14 minutes; this meets the needs of flexibility.

Building upon the capabilities of the CTC, the sample period timing software of the Collect and Store Data process was designed. The software had four purposes, calculating the CTC programming values, initializing the CTC interrupts for the sampling periods, determining the interrupt service routine parameters, and shut down of the CTC interrupts. Since the calculation of CTC programming values is a math intensive effort, this task is accomplished by a PLZ routine in the Collect_Data Module (external unit). These values are passed to the Sampler Module (internal unit) where the CTC is programmed. Also inside the Sampler Module are the interrupt service routines. The interrupt service routine used for short sampling periods employes the CTC exclusively. The routine for longer timing periods uses a sixteen bit counter in addition to the CTC timing. In both routines, a channel selection byte is written to the AIO board to initiate each analog to digital conversion. The final CTC related software accomplishes the shut down of the interrupts. These shut down activities are also in the Sampler Module portion of the software.

This division of activity between the Collect_Data Module and the Sampler Module tracks with the division of function between the internal data collection/storage unit and the external control/long term storage unit. The programming values needed by the internal unit (Sampler Module) are developed in the external unit (Collect_Data Module) and passed to the internal unit (Sampler Module) to program the data collection. Thus the software developed in PLZ for the Collect_Data Module and in assembly language for the Sampler Module reflects the dual-unit hardware concept of the data collection system.

Archival Storage of Data The final purpose of the Collect and Store Data process is the transfer of data from its temporary storage in memory into a more permanent storage. As with the previous two discussions, the capabilities available in the MCZ development system formed the basis for the design. The Zilog system's RIO Operating System supports disk file operations. It was pointless to reinvent the wheel so the RIO disk file operations became the basis for the long term data storage. In the PLZ language Collect_Data Module, a disk file is created, filled with the data from memory, and then closed. To satisfy the requirements for data integrity, a block of header information is loaded into the beginning of the raw data file. This header information holds a test identifier, a tag which all subsequent files based on this original file will also have. This tag is ment to ensure data traceability.

The activities of the Collect and Store Data process are thus implemented by the Collect_Data Module and the subordinate Sampler Module. The combination of the two modules represents the full implementation of the Collect and Store Data process, a process that involves both the internal and external units of the target data collection system. Though the Collect_Data Module is subordinate to the common user interface process, some portions of the common user interface are implemented in Collect_Data Module. Collect_Data Module sends messages to the user and performs error checking on the user supplied input parameters. Sampler Module also has one direct tie to the user, a request for a begin data collection. This was ment to simulate a trigger signal.

Set Up Scale Factor File

This process precedes the scaling of the raw data and focuses on user input of the needed scale factors. Though interaction with the user via prompts for information on the system screen and keyboard input of data, a file of scale factors is created. The scale factor file holds sixteen records, one for each of the input channels of the AIO board. The user interface is menu driven, offering the user a choice of six activities associated with editing the sixteen records of the scale factor file. The process was implemented in the Scale_Factor Module. This PLZ software was successfully complied but due to time constraints it was not integrated in with the other software. The listing of Scale_Factor Module is in Appendix B.

Scale Data

The purpose of this process is to translate the twelve bit, two's complement representations of the raw data file into scaled data. In its simplest form this would be accomplished by multiplying each channel's data by the appropriate scale factor from a scale factor file. This process was not implemented.

Output Data

This process simply prints out the contents of a data file. The header information in each file would give full identification of the original test from which the data was collected. Similarly the channel number, sampling period, number of samples, and user comments would be displayed along with the data. This process was not implemented.

User Data Manipulations

The final process is left up to the user's needs. However, to maintain data integrity, file access routines which included the necessary checks and prohibitions would be provided to the user. With these routines, the header information maintained in each file would also be maintained in any files created by user activities. These process support routines were not implemented.

Summary

In summary, the data collection system was partially implemented on a Zilog MCZ Z-8Ø development system. The data collection system was designed around five processes and a common user interface. The functions of the internal data storage unit were implemented in the assembly language Sampler Module. Some of the functions of the external data storage and control unit were implemented in the PLZ language Collect_Data Module and its subordinate Sampler Module. These implementations focus on the Collect and Store Data process. Of the remaining processes, only Set Up Scale Factor File was worked on, it being implemented in the Scale_Factor Module.

Overview of the Rest of the Thesis

The remainder of this thesis is devoted to describing the software modules. The modules are presented in a bottom up order. The modules' names and purposes are listed below along with the page numbers for the beginning of their descriptions. The listings of module software are in the appendices.

Module Name	Page	Description & Purpose
Enhancements	20	Enhancements Module is a set of PLZ language routines which make input and output in PLZ programs easier. The 38 routines are divided into three groups. There are 2Ø "write" routines, 8 "read" routines, and 1Ø internal support routines. Enhancements Module calls routines of the PLZ.STREAM.IO Module.
Utility	124	Utility Module is a collection nine assembly language routines which give PLZ language routines direct access to IO ports, memory locations, the Z-8Ø interrupts, the system data, and the operating system memory manager. To the calling PLZ program, these assembly language routines look just like PLZ subroutines.
Sampler	159	Sampler Module is a single assembly language program which sets up and executes an interrupt paced analog to digital conversion data collection system. Sampler Module supports the PLZ subroutine call structures.
Buffers	208	Buffers Module contains no code. It defines a 2,ØØØ byte memory buffer used by the data collection system.
Collect_Data	209	Collect_Data Module is a PLZ language program that controls Sampler Module's collection of data and then loads that data into a disk file. Collect_Data must be linked with the Enhancements, Sampler, and PLZ.-STREAM.IO Modules. Collect_Data Module has not been compiled.
AIO.PLZ.S	390	AIO.PLZ.S Module is a collection of PLZ language routines which, through Utility Module routines, control the AIO analog input output board of the MCZ development system. These routines were written principally as design routines; assembly language versions are in Sampler Module.
Scale_Factor	416	Scale_Factor Module is a PLZ language program through which the user would set up or edit a file of scale factors. The scale factors are used to convert raw data files into scaled data files.

II. Enhancements Module

Introduction to Enhancements Module

Enhancements Module is a collection of 38 PLZ language routines whose purpose is to make PLZ input/output more Pascal-like. The 2Ø "Write" routines and the 8 "Read" routines were written to emulate their Pascal name-sakes. Internal to the module are 1Ø support routines used for data formating, translation, and error checking. The routines are:

Internal Procedures	Write Procedures	Read Procedures
ASCII	WRITE	READLN
VALUE	WRITELN	READ_HBYTE
VALUE_LOOP	WRITE_DBYTE	READ_DBYTE
PUTCH	WRITELN_DBYTE	READ_BBYTE
GETCH	WRITE_HBYTE	READ_LBYTE
GET_ASCII_CH	WRITELN_HBYTE	READ_DINTEGER
PLACE_LOOP	WRITE_BBYTE	READ_HWORD
VALID_BINARY_CH	WRITELN_BBYTE	READ_DWORD
VALID_DECIMAL_CH	WRITE_LBYTE	
VALID_HEX_CH	WRITELN_LBYTE	
	WRITE_DINTEGER	
	WRITELN_DINTEGER	
	WRITE_DWORD	
	WRITELN_DWORD	
	WRITE_HWORD	
	WRITELN_HWORD	
	WRITE_POINTER	
	WRITELN_POINTER	
	WRITE_RCODE	
	WRITELN_RCODE	

The Enhancements Module routines were written to speed up development of other PLZ software, to make PLZ a slightly higher level language. Input/output (IO) in PLZ is somewhat cumbersome. For example, to output the string "I Like Pascal Best" using PLZ IO the statement would be:

```
RETURN_BYTES, RETURN_CODE :=
                    PUTSEQ( LOGICAL_UNIT, ^STRING, LENGTH )
```

where LOGICAL_UNIT is the logical unit number of the desired output device, ^STRING is a pointer to the string "I Like Pascal Best" ("#'I Like Pascal Best %R'" could also be used in place of "^STRING"), and LENGTH is the number of characters to be output. Thus, unlike Pascal's single input parameter for WRITELN, PUTSEQ requires three input parameters. Also unlike the Pascal WRITELN statement, this PLZ output has two output parameters. RETURN_BYTES is the number of character actually output and RETURN_ CODE is the operating system condition or error code. In contrast, using the Enhancements Module WRITELN procedure the line is:

WRITELN(LOGICAL_UNIT, #'I Like Pascal Best %R')

which has only two input parameters and no output parameters. This is possible because the Enhancements Module includes the procedures necessary to check and format the input, eliminating the need for the extra parameters. The key difference between Pascal's WRITELN and the Enhancement's Module WRITELN is the manditory inclusion of the logical unit input parameter. In Pascal, the output device number is an optional parameter.

The logical unit parameter was included for three principal reasons. First, the Enhancements routines are compiled appendages to the PLZ language, not extensions to it. Within these constraints, it simply wasn't possible to implement an optional parameter. An alternative to an optional parameter would be a output device selection function. This was rejected in lue of the logical unit parameter since one, Pascal doesn't have such a function, and two, it would increase the overhead of the Enhancements Module, including the addition of module level variables. The third reason for the inclusion of the logical unit parameter was the anticipation that many devices would be used rendering the parameter particularly useful. For these reasons, the routines of the Enhancements Module include the logical unit parameter.

The other major deviation from Pascal is the use of many read and write statements rather than just four. This was forced by the appendage nature of the Enhancements Module routines, the limitations of PLZ, and a desire to reduce the overhead for calling routines. In Pascal, the output string is parsed during compillation; PLZ does not support such actions during compiling. In Pascal, variables are converted to or from ASCII by the read and write statements; PLZ does not support such conversions. In Pascal, all output is either decimal representations or strings of ASCII characters. To input or output values in other than decimal representations requires the Pascal program to perform the conversion. By having separate routines already set up for IO in , character hexidecimal, decimal, binary, and logical formats, the burden on the calling PLZ routine is reduced. Given the nature of the Enhancements Module, the restrictions of PLZ, and the desire to reduce the overhead of calling routines, separate routines were written for each type of PLZ variable.

The Enhancements Module routines, as appendages to PLZ, do use two of the PLZ input output routines of the PLZ STREAM.IO Module (Ref 6: Sec

6). These routines, PUTSEQ and GETSEQ, are the primitive input and output routines upon which the Enhancements Module routines are built. PUTSEQ and GETSEQ are declaired external to the Enhancements Module. Their relationship to the Enhancements Module routines is shown in Figure 4.

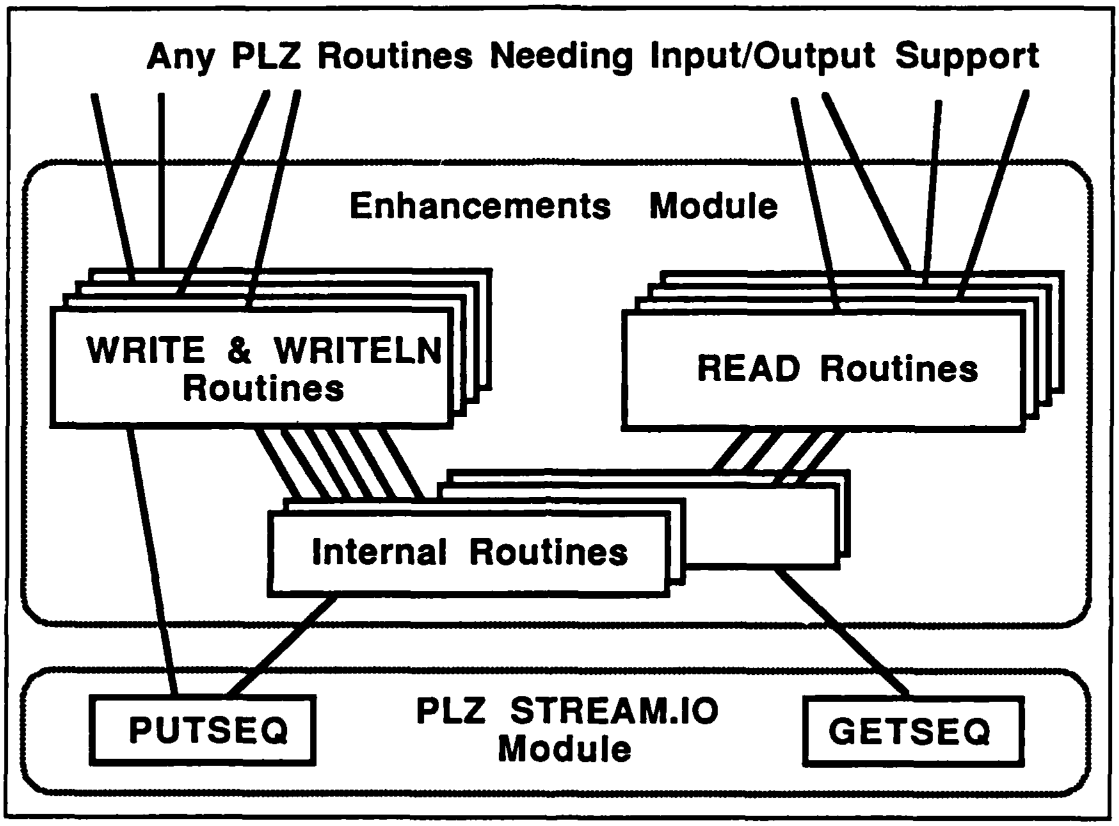


Figure 4. Relationship of Enhancements Module Routines to Calling PLZ Routines and to PLZ STREAM.IO Module Routines.

To show how the Enhancements Module routines can be used, the following are some examples of Pascal IO statements and their PLZ/Enhancements Module parallels. Carriage returns in the output are shown by "«". "%R" is the PLZ constant for a carriage return.

Example 1

Pascal:	WRITELN('This is a Text String Output');
Output:	This is a Text String Output«
PLZ:	WRITELN(PRINTER, #'This is a Text String Output %R')
Output:	This is a Text String Output«

Example 2

```
Pascal:   WRITE( CURRENT_COUNT, ' items have been sorted.' );
Output:   27 items have been sorted.
PLZ:      WRITE_DBYTE( PRINTER, CURRENT_COUNT )
          WRITE( PRINTER, #'  items have been sorted. %R' )
Output:   27. items have been sorted.
```

Example 3

```
Pascal:
     WRITELN( DIMES, ' dimes plus ', NICKELS, ' nickels totals ',TOTAL );
Output:   17 dimes plus 8 nickels totals 25«
PLZ:      WRITE_DBYTE( PRINTER, DIMES )
          WRITE( PRINTER, #' dimes plus %R' )
          WRITE_DBYTE( PRINTER, NICKELS )
          WRITE( PRINTER, #' nickels totals %R' )
          WRITELN_DBYTE( PRINTER, TOTAL )
Output:   17. dimes plus 8. nickels totals 25.«
```

Example 4

Pascal: This would require a 25+ line routine, including a 16 item case statement, to translate the decimal variables into hex.

```
PLZ:      WRITELN_HWORD( PRINTER, ADDRESS1 )
Output:   2FC7h
```

Error checking is both accomplished and ignored in Enhancements Module routines. The error checking that is performed is distributed among the routines. Gross errors, like an operating system return code for an IO error, are not passed back. Errors like these are ignored or "patched" to permit continued program operation. This approach was selected to permit the programs to stumble along rather than fatally fail during debugging. This way debugging can proceed more readily using the expected output and the debugging aid of WRITE_RCODE to figure out what went wrong. This approach is based on the belief that once final version software was reached it would be error free and diagnostic error checking would not be needed. Defensive error checking, such as GET_ASCII_CH's acceptance of only ASCII characters, remains in place.

To give an example of the distributed error checking, the figure and text below describethe process of reading in a decimal value and then outputing it as a hexidecimal value. This process involves thirteen routines, seven for input and six for output. The routines involved and their relationship is shown in Figure 5 . Error checking and ignoring is scattered throughout the thirteen routines. The following is a list of the error related actions.

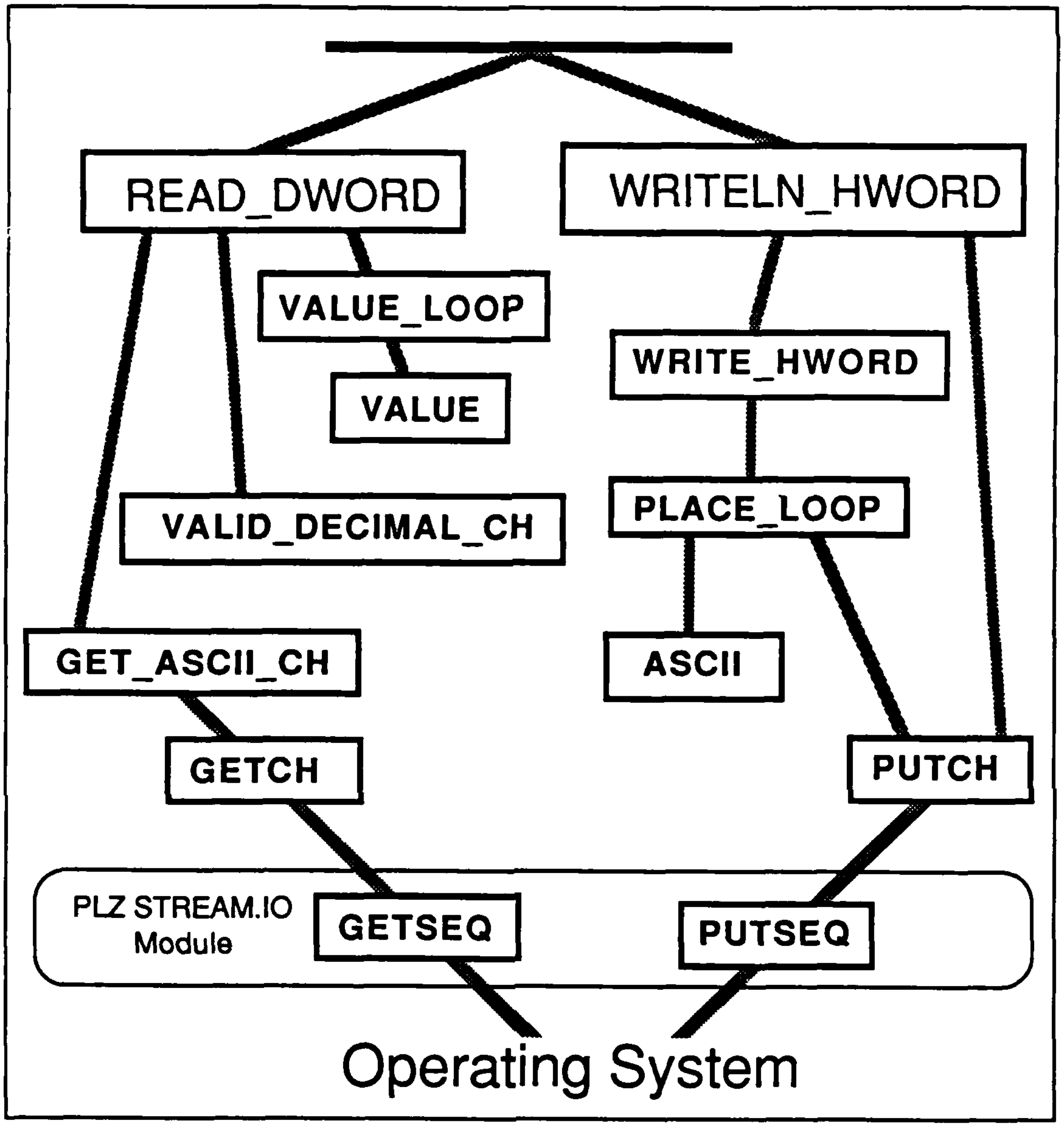


Figure 5. Routines and Relationships Used to Read in a Decimal Value and Output a Hexidecimal Value.

1. GETSEQ This is an external routine of the PLZ STREAM.IO Module. It returns to a calling routine the RIO operating system error code, RETURN_CODE, and the number of characters actually read in, LENGTH.

2. GETCH This routine calls GETSEQ to read in only one character. GETCH then ignores the return parameter LENGTH since only

3. GET_ASCII_CH Only ASCII characters are returned to the calling routine by GET_ASCII_CH. It checks the character it gets from GETCH to see whether it is a valid ASCII character. If it is, the character is returned to the calling routine. If not, GET_ASCII_CH calls GETCH for another character and keeps checking and callin GETCH until a valid ASCII character is read.

4. READ_DWORD This routine does no error checking itself. It depends upon GET_ ASCII_CH to pass only valid ASCII characters and upon VALID_DECIMAL_CH to ok only "Ø" through "9". READ_ DWORD sits in a loop, calling GET_ASCII_CH and VALID_ DECIMAL_CH until sufficient characters are input. Then READ_ DWORD depends upon VALUE_ LOOP to correctly translate the characters, all already verified as decimal, into the NUMBER passed back to the calling routine.

5. VALID_DECIMAL_CH VALID_DECIMAL_CH examines the characters passed to it. If the character is a "Ø" through "9" VALID_ DECIMAL_CH returns as TRUE, otherwise it returns as FALSE.

6. VALUE This routine is used by VALUE_LOOP to translate a character into the MAGNITUDE it represents. VALUE will translate the characters "Ø" through "F" into values of Ø through 16. If VALUE does receive a character other than these defined, it returns a MAGNITUDE of zero.

7. VALUE_LOOP With the MAGNITUDEs returned from VALUE, VALUE_ LOOP translates the string of characters into a single MAGNITUDE. VALUE_ LOOP checks for overflow with the addition of each character's contribution to the total value. If overflow is detected, the output MAGNITUDE is set to the maximum possible for a PLZ word, 65535 decimal.

At this point, READ_DWORD returns NUMBER to its calling routine. For this example, NUMBER is immediately passed to WRITELN_HWORD.

8. WRITELN_HWORD This routine depends upon WRITE_HWORD and PUTCH to handle errors and expects its calling routine to pass only valid NUMBERs to be output.

9. WRITE_HWORD This routine does no error checking. It depends upon PLACE_ LOOP to translate NUMBER into ASCII characters and PUTCH to output the "h". It also expects its calling routine to pass only valid NUMBERs.

10. PLACE_LOOP This routine also does no error checking. It breaks down

the NUMBER, from most significant place to the ones place, determining the VALUE of each place. PLACE_LOOP depends upon ASCII to correctly translate the VALUEs into ASCII characters and upon PUTCH to output those characters.

11. ASCII — Through the use of a case statement ASCII translates VALUEs into CHARACTERs. If the value passed exceeds 16 decimal, ASCII returns a blank. Thus if any of the higher lever routines errored, ASCII will return either a blank or an erroneous character between "Ø" and "F". Thus, the program will continue to execute though flawed output may occur.

12. PUTCH — This routine ignores all errors returned by PUTSEQ. PUTCH calls PUTSEQ to output only one character. PUTCH assumes that the single character is successfully output. PUTCH also ignores the PUTSEQ output parameter RETURN_CODE assuming that the output was successful. This permits the program to continue execution.

13. PUTSEQ — This is an external routine of the PLZ STREAM.IO Module. Its error checking has two return parameters, the RIO operating system RETURN_CODE, and the number of characters actually output, LENGTH.

While Enhancements Module is a complete set of IO support routines intended to ease the IO programming in PLZ, not all PLZ applications will require all of the routines. In these cases, a new module, containing only the needed routines could be formed and linked in with the application program. An example of such a module, DEBUGS, is listed in Appendix A. Alternatively, the Enhancements routines needed could be part of the calling routine's module. An example of this approach is Scale_Factor Module (Appendix H). In either case, the PLZ STREAM.IO must be linked in for access to PITSEQ and GETSEQ.

If speed of execution is of concern, the overhead of the Enhancements Module routines could be reduced by combining the code of several routines into one larger routine. This would eliminate the overhead and delay of subroutine calls present in the current set of routines. For example the six routines used in the example above to read in a decimal value could be combined into a single routine version of READ_DWORD. The negative impact of this approach would be the duplication of many lines of code in the combined routines.

In conclusion, the 38 PLZ routines of the Enhancements Module were written to make IO in PLZ a little easier, in effect to make PLZ a slightly higher level language. These routines have defensive error checking distributed throughout the routines but patch or ignore fatal errors in the belief that a routine that stumbles along is easier to debug than one which fails completely. Though the Enhancements Module is a complete set of IO routines, not all applications will require all 38 routines. In these cases, a module of selected routines could be used or the routines needed could be put into the application program's module. In either case the PLZ STREAM.IO Module must be linked in.

The following pages detail the 38 Enhancements Module routines. For each routine the documentation includes:

1. Name of the routine or routines,
2. Name of module,
3. Language routine is witten in and number of lines of code,
4. A synopsis of the routine or routines,
5. A data flow diagram showing the relationship of the routine to its calling routines and to routines it calls,
6. How the routine is invoked including the input parameter passing schema and a list of the routines which call,
7. The variables and constants used by the routine at the global, module, and routine level,
8. The names, purpose, invocation, and parameter passing of any other routines called by the routine,
9. The output of the routine and any system configuration changes produced by the routine,
10. The testing of the routine and the results of the testing, and
11. The location of the program listing.

The program listings for Enhancements Module and the various test routines are in Appendix A. Further information on the PLZ language can be found in references five and six.

1. Name of Routine: **ASCII**

2. Internal routine of Enhancements Module.

3. Written in PLZ; 22 lines of executable code.

4. Synopsis of Routine

ASCII is an internal support routine of the Enhancements Module. It translates a hexadecimal value (Ø through F) into the ASCII character which represents that value ("Ø" through "F"). To facilitate the use of leading blanks in stings of values, ASCII will return a blank (ASCII 2Ø hex) rather than a zero (ASCII 30 hex) if blanking is selected.

5. Routine Relationship Diagram

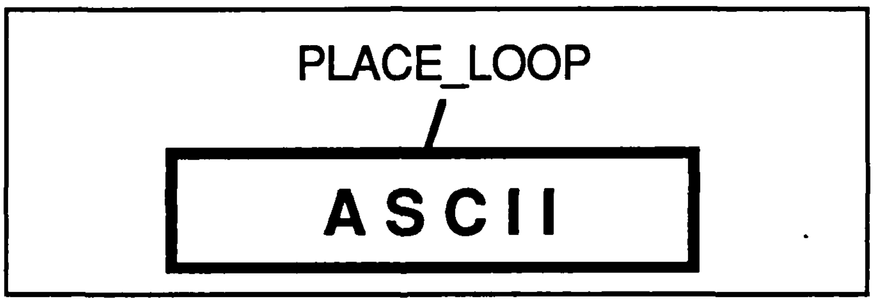


Figure 6. Relationship of ASCII to PLACE_LOOP.

6. Invocation

a. Invocation Statement

ASCII is invoked by:

OUT_BLANKING, CHARACTER := ASCII (VALUE, IN_BLANKING)

b. Parameter Passing Schema

ASCII has two input parameters, VALUE, type Word, and IN_BLANKING, type Byte. VALUE is the hexadecimal quantity that is to be translated into the correct ASCII character. IN_BLANKING is a logical parameter which indicates wether values of zero should be returned as a "Ø", when IN_BLANKING is false, or as a blank, when IN_BLANKING is true. The output parameters are discussed below.

c. Routines Which Call ASCII

ASCII is an internal support routine for Enhancements Module was written to be called only by PLACE_LOOP, another internal routine of Enhancements Module.

7. Variables and Constants

a. Global

ASCII uses no globally defined constants or variables.

b. Module

ASCII uses three Enhancements Module constants:
TRUE: Value of 1, logical true,
FALSE: Value of Ø, logical false, and
BLANK: Value 2Ø hex, ASCII blank character.

8. Other Routines Called

ASCII calls no other routines.

9. Output of Routine

a. Parameter Passing Schema

ASCII has two output parameters, CHARACTER and OUT_ BLANKING, both of type Byte. CHARACTER is returned as the ASCII character which represents the VALUE input to the routine. However, if IN_BLANKING was True and VALUE was zero, CHARACTER will be returned as a blank (ASCII 20 Hex). OUT_BLANKING is a logical parameter, true if CHARACTER is returned as a blank, false otherwise. OUT_BLANKING is a flag to the calling routine that a blank was returned.

b. System Configuration Changes

ASCII causes no system changes.

10. Routine Testing

a. Description of Test

ASCII was tested in combination with its calling routine (PLACE_LOOP) and the hexidecimal output routines WRITE_HBYTE and WRITELN_HBYTE. The PLZ program output hexadecimal characters to the system console. Out of range and undefined values were used in addition to a range of valid values. Unless all routines were working, no output would occur.

b. Results of Test

The proper characters were output to the system console for all cases tested.

11. Reference to Listing

ASCII's listing is on page 280 in Appendix A.

1. Routine Name: **VALUE**

2. Part of Enhancements Module

3. Written in PLZ. 19 lines of executable code.

4. Synopsis of Routine

VALUE is an internal support routine of the Enhancements Module. It is used to convert from ASCII characters (Ø to 9 and A to F) into their hexadecimal values. If an undefined character is passed, a value of Ø hex is returned. VALUE supports some of the READ statements of the Enhancements Module.

5. Routine Relationships Diagram

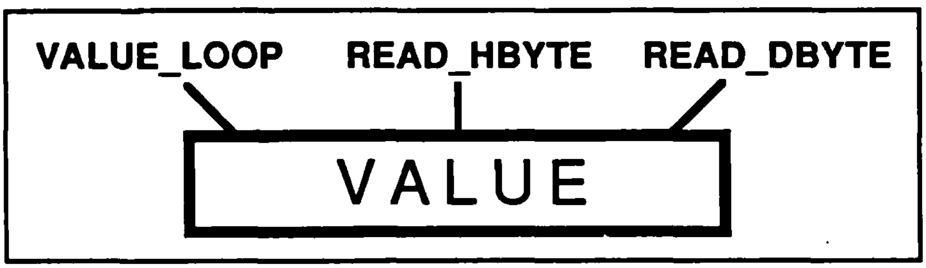


Figure 7. Relationship of VALUE to Other Routines.

6. Invocation

a. Invocation Statement

VALUE is invoked via:

MAGNITUDE := VALUE(CHARACTER)

where CHARACTER and MAGNITUDE are both of type Byte.

b. Parameter Passing Schema

VALUE has one input parameter, CHARACTER, the ASCII character that is to be translated into a hexidecimal value.

c. Routines Which Call VALUE.

VALUE is an internal support routine of the Enhancements Module. It was written to be called only by VALUE_LOOP, READ_HBYTE, and READ_DBYTE.

7. Variables and Constants

VALUE uses no constants or variables outside of its input and output parameters.

8. Other Routines Called

VALUE calls no other routines.

9. Output of Routine

a. Parameter Passing Schema

VALUE has a single output parameter, MAGNITUDE, the hexidecimal value represented by the input parameter CHARACTER.

b. System Configuration Changes

VALUE causes no system configuration changes.

10. Routine Testing

a. Description of Test

VALUE was tested in concert with VALUE_LOOP, READ_HBYTE, and READ_DBYTE. A short PLZ program read in values from the keyboard and output their value to the system console. Out of range and undefined values were also input. Unless all the routines worked, proper output would not occur.

b. Results of Test

The correct values were output including when improper values were input.

11. Reference to Listing

VALUE's listing is on page 281 in Appendix A.

1. Routine Name: **VALUE_LOOP**

2. Internal routine of Enhancements Module.

3. Written in PLZ; 11 lines of executable code.

4. Synopsis of Routine

VALUE_LOOP is an internal support routine of the Enhancements Module; it is used by some of the READ routines. VALUE_LOOP translates a string of ASCII characters into the value they represent. The string of ASCII characters (1 to 8 characters) can be in any base as the base is input to VALUE_LOOP. The routine translates each character into a value (via routine VALUE), multiplies that value by the base factor for that character's position, and then adds the character's full value to the cumulative value. This process begins with the least significant bit and proceeds through the higher significence bits. If the translated value exceeds the maximum value for a PLZ word (65535 decimal) the output value is set to the maximum. The routine ends when a blank is detected or when eight characters have been translated.

5. Routine Relationship Diagram

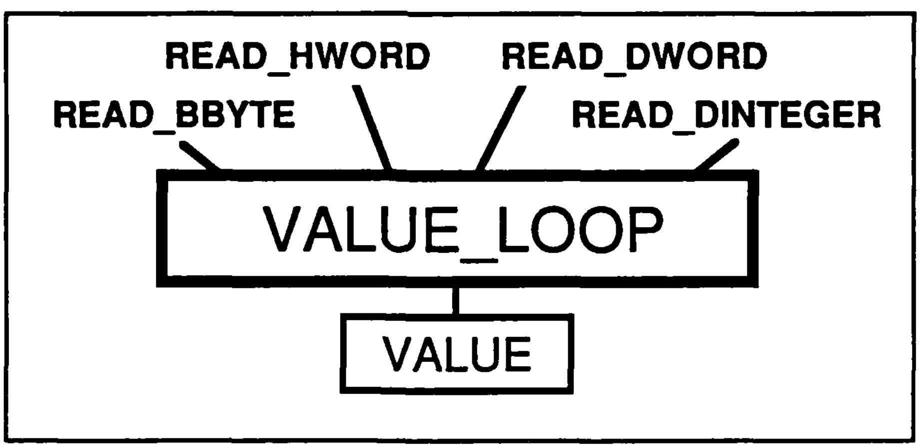


Figure 8. Relationship of VALUE_LOOP to Other Routines.

6. Invocation

a. Invocation Statement

VALUE_LOOP is called via:

MAGNITUDE := VALUE_LOOP(INPUT_STRING, MULTIPLIER)

where MAGNITUDE and MULTIPLER are of type Word and INPUT_STRING is a pointer to an ASCII string.

b. Input Parameter Passing Schema

VALUE_LOOP has two input parameters. INPUT_STRING is a pointer to the string of ASCII characters to be translated. MULTIPLER is the base of the number represented by the string of characters. As it is type Word it has a defined range of Ø to 65535 decimal though its useful range is 2 to 16 decimal.

c. Routines Which Call VALUE_LOOP

VALUE_LOOP is an internal support routine of the Enhancements Module. It was written to be called only by READ_BBYTE, READ_DINTEGER, READ_HWORD, and READ_DWORD. This is important as error checking is distributed among the routines.

7. Variables and Constants

a. Global

VALUE_LOOP uses no module level variables or constants.

b. Module

VALUE_LOOP uses one module level constant, BLANK: The ASCII value 20 hex for a blank. VALUE_LOOP uses no module level variables.

c. Routine

VALUE_LOOP uses two routine level variables, INDEX and FACTOR. INDEX, type byte, is used to advance through the input character string. Its initial value is zero. FACTOR, type word, holds the base value of the current character position. It is the base (MAGNITUDE) raised to the INDEX power. Its initial value is one.

8. Other Routines Called by VALUE_LOOP

VALUE_LOOP calls procedure VALUE to translate each character into the value it represents. VALUE is also an internal support routine of Enhancements Module. VALUE is invoked via:

MAGNITUDE := VALUE(CHARACTER)

where CHARACTER is the ASCII character to be converted into the MAGNITUDE it represents. Both CHARACTER and MAGNITUDE are of type Byte.

9. Output of Routine

a. Output Parameter Passing Schema

VALUE_LOOP has one output parameter, MAGNITUDE, of type Word. It is the value represented by the input character string of the input base. MAGNITUDE can take on a value of Ø to 65535 decimal. If the value of the input string exceeds the maximum value, the maximum value will be returned.

b. System Configuration Changes

VALUE_LOOP causes no system configuration changes.

10. Routine Testing

a. Description of Test

VALUE_LOOP was tested along with other Enhancements Module routines. The complete set of routines are necessary for correct function. The integrating PLZ program read in character strings from the keyboard, translated their value (using VALUE_LOOP and VALUE), and then output the value to the system console. Various valid character strings and several out of range and invalid strings were input.

b. Results of Test

The correct value was output to the console for all cases tested.

11. Reference to Listing

The listing of VALUE_LOOP is on page 282 in Appendix A.

1. Routine Name: **PUTCH**

2. Internal routine of Enhancements Module.

3. Written in PLZ; three executable lines of code.

4. Synopsis of Routine

PUTCH is an extremely short routine which interfaces the output routines of Enhancements Module with the output routine of the PLZ Stream IO Module, PUTSEQ. Where PUTSEQ has five parameters (three input and two output), PUTCH as only two input parameters. PUTCH thus insulates the output routines of the Enhancements Module from the added complexities of PUTSEQ. PUTCH is based on a sample routine given the the PLZ Documentation (Ref 6: 6-5).

5. Routine Relationships Diagram

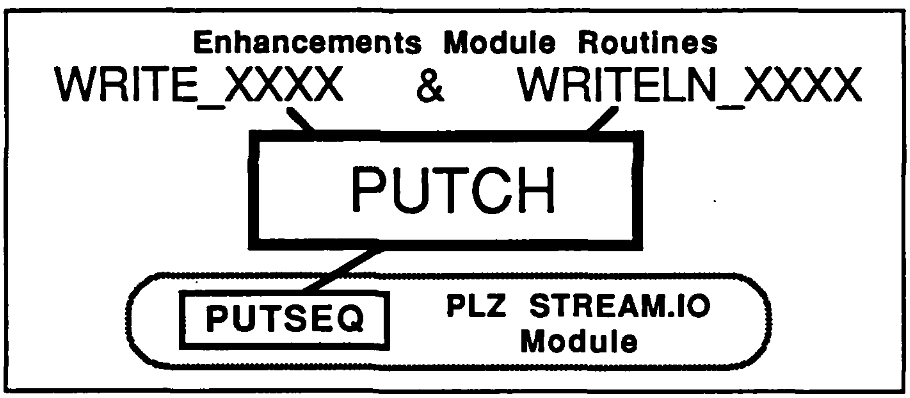


Figure 9. Relationship of PUTCH to Other Routines.

6. Invocation

a. Invocation Statement

PUTSEQ is invoked as follows.

PUTCH(LOGICAL_UNIT, CHARACTER)

where both input parameters are of type Byte.

b. Parameter Passing Schema

PUTCH has two input parameters. LOGICAL_UNIT is the number of the device the output is to be routed to. CHARACTER is the value to be output to the desired LOGICAL_UNIT. Though its name implies ASCII data, any eight bit hexidecimal value can be passed though CHARACTER.

c. Routines Which Call PUTCH.

PUTCH is an internal support routine of the Enhancements Module. It was written to be called only by Enhancement Module routines. PUTCH is called by PLACE_LOOP, WRITELN, WRITE_DBYTE, WRITELN_DBYTE, WRITE_HBYTE, WRITELN_HBYTE, WRITE_BBYTE, WRITELN_BBYTE, WRITELN_LBYTE, WRITE_DINTEGER, WRITELN_DINTEGER, WRITE_DWORD, WRITELN_DWORD, WRITE_HWORD, WRITELN_HWORD, and WRITELN_POINTER.

7. Variables and Constants

a. Global

PUTCH uses no global variables or constants.

b. Module Level

PUTCH uses no module level variables or constants.

c. Routine Level

Within the routine are two variables, LENGTH (type Word) and RETURN_CODE (type Byte). LENGTH is used as both for input and output parameters to the external routine PUTSEQ. For input it is set to one as PUTCH outputs only one byte to PUTSEQ. LENGTH is used as a place keeper output variable – there only to keep the subroutine calling syntax correct. RETURN_CODE is similarly used as a place keeper output parameter.

8. Other Routines Called

PUTCH calls PUTSEQ, an external routine of the PLZ STREAM IO Module. PUTSEQ outputs a known length sequence of values to the specified logical unit. PUTSEQ is invoked by:

```
LENGTH, RETURN_CODE :=
                PUTSEQ( LOGICAL_UNIT, BUFFER_PTR, LENGTH )
```

PUTSEQ has three input parameters, LOGICAL_UNIT, BUFFER_PTR, and LENGTH. LOGICAL_UNIT (type Byte) is the number of the device to which data is to be output. BUFFER_PTR (type Pointer to Byte) is a pointer to the string of characters (or values) to be output to the designated logical unit. Note that as PUTCH outputs only single characters, BUFFER_PTR is passed as pointer to the PUTCH input parameter CHARACTER. Thus, in the call to PUTSEQ, CHARACTER undergoes a type conversion from Byte to Pointer-to-Byte. The third input parameter, LENGTH (type Word), is the number of characters (values) to be output; the length of the string pointed to by BUFFER_PTR. The PUTSEQ call in PUTCH uses the constant one for LENGTH as only a single character is output.

PUTSEQ returns two parameters, LENGTH and RETURN_CODE. LENGTH (type Word) is the number of bytes actually output. RETURN_CODE is the operating system error code.

9. Output of Routine

PUTCH has no output parameters. Beyond writing a value to a logical unit, PUTCH has no impact on system configuration.

10. Routine Testing

PUTCH was not specifically tested. Rather, it was tested along with the other routines of the Enhancements Module. Most of the "write" and "writeln" routines use PUTCH, directly or indirectly. These routines worked, thus PUTCH worked.

11. Reference to Listing

The listing for PUTCH is on page 283 in Appendix A.

1. Routine Name: **GETCH**

2. Internal routine of Enhancements Module.

3. Written in PLZ; four lines of executable code.

4. Synopsis of Routine

GETCH is a very simple routine which interfaces the PLZ STREAM IO Module routine GETSEQ to the "read" routines of the Enhancements Module. Where GETSEQ has three input parameters and two output paramenters, GETCH presents its calling PLZ routine with one input and one output parameter. The key difference is that GETSEQ can read in a string of arbitrary length while GETCH reads in a single value. GETCH is based on a sample routine given the the PLZ Documentation. (Ref 6: 6-5)

5. Routine Relationships Diagram

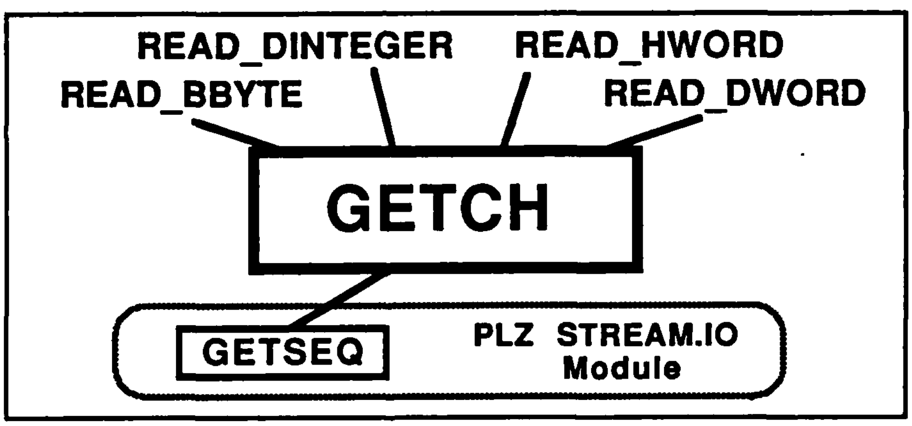


Figure 10. Relationship of GETCH to Other Routines

6. Invocation

a. Invocation Statement

GETCH is invoked by:

CHARACTER := GETCH(LOGICAL_UNIT)

where both CHARACTER and LOGICAL_UNIT are of type Byte.

b. Parameter Passing Schema

GETCH uses the input parameter LOGICAL_UNIT to select the device from which a value is to be read. The value read is output via parameter CHARACTER. Despite its name, CHARACTER could output any eight bit value.

c. Routines Which Call GETCH.

GETCH is an internal routine of the Enhancements Module and was written to be called only by other Enhancements Module routines. GETCH is called by GET_ASCII_CH.

7. Variables and Constants

a. Global Level

GETCH uses no global variables or constants.

b. Module Level

GETCH uses no module level variables. It does use two module level constants, OPERATION_OK and BLANK. OPERATION_OK is the operating system return code for a successful IO action; its value is 8Ø hexidecimal. BLANK is the ASCII blank, value 2Ø hexidecimal.

c. Variables and Constants internal to GETCH

GETCH has two internal variables and one internal constant. The internal variables, RETURN_CODE (type Byte) and LENGTH (type Word), are used in calling GETSEQ. The constant used, 1, is explicit (not a named constant) and is also used in calling GETSEQ.

8. Other Routines Called

GETCH calls a single routine, GETSEQ, an external routine of the PLZ STREAM.IO Module. GETCH uses GETSEQ to read a single character from a designated logical unit. GETSEQ has one input parameter, LOGICAL_UNIT (type Byte); one return parameter, RETURN_CODE (type Byte); and two bidirectional parameters, BUFFER_PTR (type Pointer-to-Byte) and LENGTH (type Word). LOGICAL_UNIT passes the number of the device driver from which the character will be taken. This is the same as the LOGICAL_UNIT passed into

GETCH. RETURN_CODE carries back the operating system code indicating whether the input was successful or not. If RETURN_CODE does not pass back the OPERATION_OK code, GETCH returns to its calling routine a blank.

BUFFER_PTR points the the memory location where the first character of the string will be stored. Thus, it is similar in function but different in type from the GETCH input parameter CHARACTER. In the invocation of of GETSEQ, BUFFER_PTR is passed ^CHARACTER or pointer to the variable CHARACTER, type Pointer-to-Byte. In this way the type conversion occurs.

LENGTH serves two purposes. On the call to GETSEQ, LENGTH gives the number of characters which are supposed to be read in. Upon return to GETCH, LENGTH passes back the number of characters actually read. For GETCH, LENGTH is passed to GETSEQ with the constant value of 1 as a single character is to be output; the return value of LENGTH is ignored.

GETSEQ is invoked via:

```
LENGTH, RETURN_CODE :=
                GETSEQ( LOGICAL_UNIT, CHARACTER, LENGTH )
```

9. Output of Routine

GETCH returns to its calling routine a single ASCII character in the output parameter CHARACTER (type Byte). If the reading operation was unsuccessful for any reason, a blank is returned to the calling routine. Beyond reading a character in from a logical unit, GETCH causes no system configuration changes.

10. Routine Testing

a. Description of Test

GETCH was tested with the rest of the Enhancements Module routines.

b. Results of Test

GETCH worked properly.

11. Reference to Listing

The listing of GETCH is on page 283 in Appendix A.

1. Routine Name: **GET_ASCII_CH**

2. Internal routine of Enhancements Module.

3. Written in PLZ; three lines of executable code.

4. Synopsis of Routine

GET_ASCII_CH reads in values from a designated logical unit and checks that the value read in is a valid ASCII character. If the value is valid, the character is returned to the calling PLZ routine. Otherwise, GET_ASCII_CH keeps reading in values until a valid character is read. The values GET_ASCII_CH considers valid are:

All printing characters: Ø-9, a-z, A-Z, and punctuation,
Control-G, the aural tone,
Control-I, horizonal tab,
Control-J, line feed,
Control-M, carriage return,
Control-[, escape, and
blank.

5. Routine Relationships Diagram

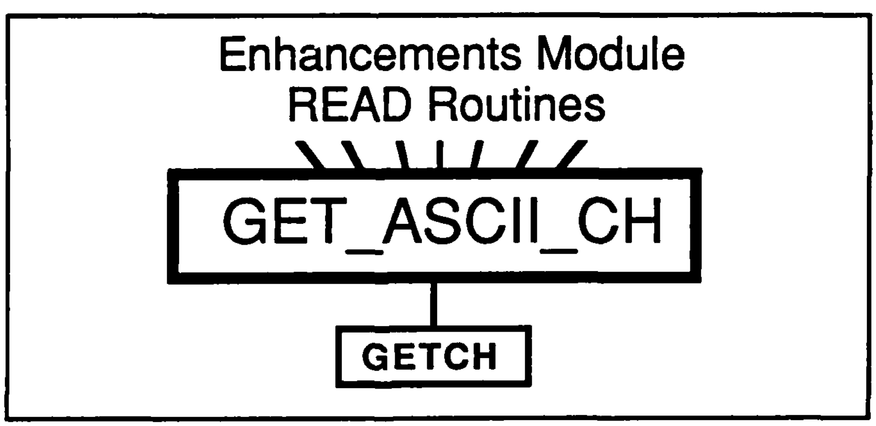


Figure 11. Relationship of GET_ASCII_CH to Other Routines.

6. Invocation

a. Invocation Statement

GET_ASCII_CH is invoked via:

CHARACTER := GET_ASCII_CH(LOGICAL_UNIT)

where both CHARACTER and LOGICAL_UNIT are of type Byte.

b. Parameter Passing Schema

The input parameter LOGICAL_UNIT is used to designate which device the value is to be read from.

c. Routines Which Call GET_ASCII_CH.

GET_ASCII_CH is an internal routine of Enhancements Module and was written to be called only by other Enhancements Module routines. GET_ASCII_CH is called by READLN, READ_HBYTE, READ_DBYTE, READ_BBYTE, READ_LBYTE, READ_DINTEGER, READ_HWORD, and READ_DWORD.

7. Variables and Constants

a. Global

Aside for the definitions for ASCII characters, GET_ASCII_CH uses no gloal variables or constants.

b. Module Level

Within the Enhancements Module, a number of constants are used to represent nonprinting ASCII characters. GET_ASCII_CH uses:

BELL:	ASCII Control-G, the aural tone,
TAB:	ASCII Control-I, horizonal tab,
LINE_FEED:	ASCII Control-J,
CARRIAGE_RETURN:	ASCII Control-M,
ESCAPE:	ASCII Control-[, and
BLANK:	ASCII for a space.

GET_ASCII_CH uses no module level variables.

c. Routine Level

GET_ASCII_CH has no routine level variables or constants.

8. Other Routines Called

GET_ASCII_CH uses another Enhancements Module routine, GETCH, to read a character from the device designated by the input parameter LOGICAL_UNIT (type Byte). If the reading operation was successful, GETCH re-turns the ASCII character in return parameter CHARACTER (type Byte). If the reading operation was unsuccessful, GETCH returns a blank. GETCH is invoked via:

CHARACTER := GETCH(LOGICAL_UNIT).

9. Output of Routine

a. Parameter Passing Schema

GET_ASCII_CH has one output parameter, CHARACTER (type Byte) which returns an ASCII character to the calling routine.

b. System Configuration Changes

Beyond reading in one or more values from the designated logical unit, GET_ASCIIC_CH causes no system configuration changes.

10. Routine Testing

a. Description of Test

GET_ASCII_CH was not tested independently. It was tested in concert with the "read" routines of the Enhancements Module. All of the read routines use GET_ASCII_CH to input characters. Thus, any test of these read routines tests GET_ASCII_CH.

b. Results of Test

The "read" routines of the Enhancements Module functioned properly. Thus GET_ASCII_CH works properly.

11. Reference to Listing

GET_ASCII_CH's program listing is on page 284 in Appendix A.

1. Routine Name: **PLACE_LOOP**

2. Internal routine of Enhancements Module.

3. Written in PLZ; seven lines of executable code.

4. Synopsis of Routine

PLACE_LOOP is an internal support routine of the Enhancements Module. It outputs to the designate device a string of ASCII characters representing the value NUMBER. The base of the output representation (defined range 2 to 16) and the number of characters output is selectable. Blanking of leading zeros is also selectable.

PLACE_LOOP works its way down from the most significant place to the least significant. At each place, $(\text{base})^P$, the contribution of NUMBER to the mantissa is found and translated into a character representing the mantissa. For example, if the base is 16 and the contribution is 11, the character would be B. NUMBER is reduced by the mantissa contribution and PLACE_LOOP proceeds to the next lower significance place. This process continues until NUMBER is completely represented.

5. Routine Relationships Diagram

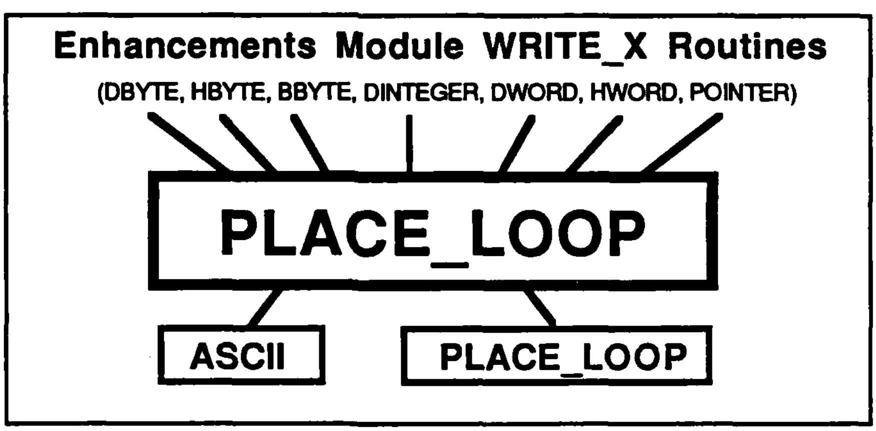


Figure 12 . Relationship of PLACE_LOOP to Other Routines.

6. Invocation

a. Invocation Statement

PLACE_LOOP is invoked by:

PLACE_LOOP(LOGICAL_UNIT, BLANKING, NUMBER, INDEX, DIVISOR)

where LOGICAL_UNIT and BLANKING are of type Byte and NUMBER, INDEX, and DIVISOR are of type Word.

b. Parameter Passing Schema

PLACE_LOOP has five input parameters. Their definitions and uses follow.

LOGICAL_UNIT:	The number of the device to which characters will be written. Type Byte.
BLANKING:	A logical flag which, if true, indicates that leading zeros are to be suppressed. Type Byte.
NUMBER:	The value which will be output. Type Word.
INDEX:	The base value of the most significant character of the output. For example, if the output is decimal with four digits, INDEX would be 1,ØØØ or $1Ø^4$. INDEX is of type Word.
DIVISOR:	The base of the output. Type Word.

c. Routines Which Call PLACE_LOOP

PLACE_LOOP is an internal routine of the Enhancements Module and was written to be called only by other Enhancements Module routines. PLACE_LOOP is called by WRITE_DBYTE, WRITE_HBYTE, WRITE_BBYTE, WRITE_DINTEGER, WRITE_DWORD, WRITE_HWORD, and WRITE_POINTER.

7. Variables and Constants

a. Global

PLACE_LOOP uses no global variables or constants.

b. Module

PLACE_LOOP uses no module level variables or constants.

c. Routine

PLACE_LOOP has two routine level variables, VALUE (type Word) and CHARACTER (type Byte) in addition to the input parameters NUMBER and INDEX. The following shows how the variables of PLACE_LOOP function to resolve the characters which represent NUMBER. Once VALUE is resolved it is translated into a CHARACTER by routine ASCII.

$$
\begin{array}{rlcl}
 & mantissa_3 \text{ X } base^3 & = & VALUE_{(3)} \text{ X } INDEX_{(3)} \\
 & mantissa_2 \text{ X } base^2 & = & VALUE_{(2)} \text{ X } INDEX_{(2)} \\
 & mantissa_1 \text{ X } base^1 & = & VALUE_{(1)} \text{ X } INDEX_{(1)} \\
+ & mantissa_{\emptyset} \text{ X } base^{\emptyset} & = & + \; VALUE_{(\emptyset)} \text{ X } INDEX_{(\emptyset)} \\
\hline
= & NUMER & = & NUMBER
\end{array}
$$

where

$VALUE_{(n)} = NUMBER_{(n)} / INDEX_{(n)}$,

$NUMBER_{(n-1)} = NUMBER_{(n)}$ **MOD** $INDEX_{(n)}$, and

$INDEX_{(n-1)} = INDEX_{(n)} / DIVISIOR$.

The calculation of VALUE and translation of the VALUEs into characters begins with the most significant position and proceeds to the least significant.

8. Other Routines Called

PLACE_LOOP calls two Enhancement Module routines ASCII and PUTCH. ASCII is used to translate the VALUEs into CHARACTERs. ASCII receives VALUE and BLANKING (passed into PLACE_LOOP by the calling routine) as input parameters and returns to PLACE_LOOP BLANKING and CHARACTER. If VALUE is zero and BLANKING is true, CHARACTER will be returned as a blank and BLANKING as turn. Otherwise, CHARACTER will be the ASCII character which represents VALUE and BLANKING will be returned as false.

PLACE_LOOP uses PUTCH to output each CHARACTER. PUTCH receives LOGICAL_UNIT (passed into PLACE_LOOP by the calling routine) and CHARACTER. PUTCH outputs the character to the desired device. PUTCH has no return parameters.

9. Output of Routine

a. Parameter Passing Schema

PLACE_LOOP has no output parameters.

b. System Configuration Changes

Other than the outputing of a string of characters to a device, PLACE_LOOP causes no system configuration changes.

10. Routine Testing

a. Description of Test

PLACE_LOOP was not individually tested. Instead it was included in a test of all the Enhancement Module routines. As many of the "write" and "writeln" routines depend upon PLACE_LOOP, if PLACE_LOOP didn't work, they wouldn't work.

b. Results of Test

The "write" routines functioned properly, thus PLACE_LOOP functioned properly.

11. Reference to Listing

The listing of PLACE_LOOP is on page 285 in Appendix A

1. Routine Name: **VALID_BINARY_CH**

2. Internal routine of Ehancements Module.

3. Written in PLZ; four lines of executable code.

4. Synopsis of Routine

VALID_BINARY_CH a simple internal support routine of the Enhancements Module. It examines an input character and determines whether it is a "Ø" or a "1". If it is, VALID_BINARY_CH returns the flag VALIDITY as true; otherwise VALIDITY is false.

5. Routine Relationships Diagram

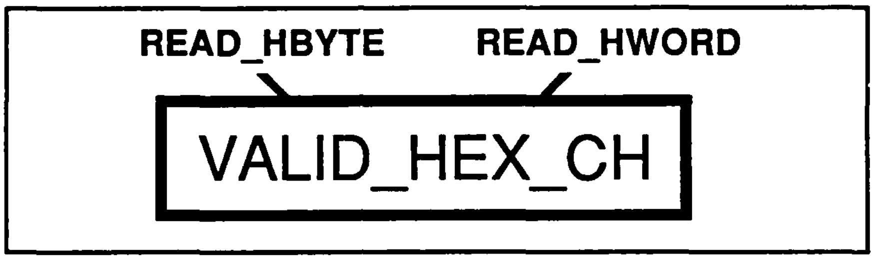


Figure 13. Relationship of VALID_BINARY_CH to Calling Routines.

6. Invocation

a. Invocation Statement

VALID_BINARY_CH is invoked via:

VALIDITY := VALID_BINARY_CH(CHARACTER)

where both VALIDITY and CHARACTER are of type Byte.

b. Parameter Passing Schema

VALID_BINARY_CH has one input and one output parameter. CHARACTER is passed into the routine and is checked against "1" and "Ø". VALIDITY is returned as either true if CHARACTER checks out. Otherwise VALIDITY is returned as false.

c. Routines Which Call

VALID_BINARY_CH is an internal routine of Enhancements Module. It was written to be called only by other Enhancements Module routines. As it turns out, VALID_ BINARY_CH is not called by any routines of the Enhancements Module. In writing the other routines, an IF statement was used to determine whether the input character was a "1" or a "Ø" rather than calling VALID_ BINARY_CH.

7. Variables and Constants

a. Global

VALID_BINARY_CH uses no global constants or variables.

b. Module Level

VALID_BINARY_CH uses two module constants: TRUE - value 1 hex, and logical true, and FALSE - value Ø hex, logical false. VALID_BINARY_CH uses no module level variables.

c. Routine

VALID_BINARY_CH hs no routine level constants or variables.

8. Other Routines Called

VALID_BINARY_CH calls no other routines

9. Output of Routine

a. Parameter Passing Schema

VALID_BINARY_CH has a single output parameter, VALIDITY, of type Byte. It is returned with the logical value true if the input CHARACTER is either a "1" or a "Ø". Otherwise VALIDITY is returned with the logical value false.

b. System Configuration Changes

VALID_BINARY_CH causes no configuration changes.

10. Routine Testing

VALID_BINARY_CH was not tested since it isn't used. However, this routine is vary similar to VALID_DECIMAL_CH and VALID_HEX_CH. These routines performed properly. Based on their similarity, it is likely that VALID_BINARY_CH would perform properly.

11. Reference to Listing

VALID_BINARY_CH's listing is on page 286 in Appendix A.

1. Routine Name: **VALID_DECIMAL_CH**

2. Internal routine of Ehancements Module.

3. Written in PLZ; four lines of executable code.

4. Synopsis of Routine

VALID_DECIMAL_CH a simple internal support routine of the Enhancements Module. It examines an input character and determines whether it is a "Ø" through "9". If it is one of these characters, VALID_DECIMAL_CH returns the flag VALIDITY as true; otherwise VALIDITY is false.

5. Routine Relationships Diagram

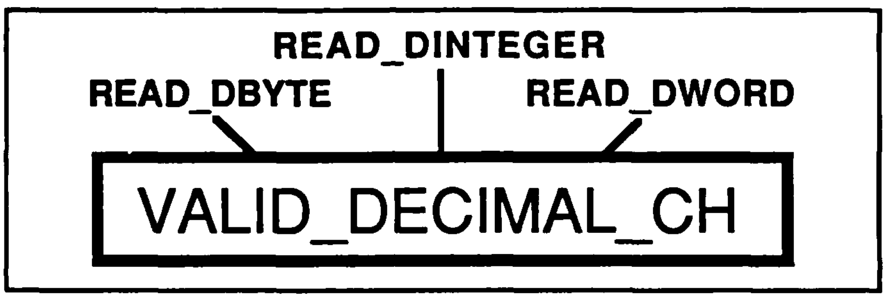


Figure 14. Relationship of VALID_DECIMAL_CH to Other Routines

6. Invocation

a. Invocation Statement

VALID_DECIMAL_CH is invoked via:

VALIDITY := VALID_DECIMAL_CH(CHARACTER)

where both VALIDITY and CHARACTER are of type Byte.

b. Parameter Passing Schema

VALID_DECIMAL_CH has one input and one output parameter. CHARACTER is passed into the routine and is checked against characters "Ø" through "9". VALIDITY is returned as either true if CHARACTER checks out. Otherwise VALIDITY is returned as false.

c. Routines Which Call

VALID_DECIMAL_CH is an internal routine of Enhancements Module. It was written to be called only by other Enhancements Module routines. VALID_DECIMAL_CH is called by READ_DBYTE, READ_DINTEGER, and READ_DWORD.

7. Variables and Constants

a. Global

VALID_DECIMAL_CH uses no global constants or variables.

b. Module Level

VALID_DECIMAL_CH uses two module constants: TRUE - value 1 hex, logical true, and FALSE - value Ø hex, logical false.

c. Routine

VALID_DECIMAL_CH hs no routine level constants or variables.

8. Other Routines Called

VALID_DECIMAL_CH calls no other routines.

9. Output of Routine

a. Parameter Passing Schema

VALID_DECIMAL_CH has a single output parameter, VALIDITY, of type Byte. It is returned with the logical value true if the input CHARACTER is a "Ø" through "9". Otherwise VALIDITY is returned with the logical value false.

b. System Configuration Changes

VALID_DECIMAL_CH causes no configuration changes.

10. Routine Testing

a. Description of Test

VALID_DECIMAL_CH was tested in conjunction with the rest of the Enhancements Module.

b. Results of Test

VALID_DECIMAL_CH works.

11. Reference to Listing

The listing for VALID_DECIMAL_CH is on page 286 in Appendix A.

1. Routine Name: **VALID_HEX_CH**

2. Internal routine of Ehancements Module.

3. Written in PLZ; four lines of executable code.

4. Synopsis of Routine

VALID_HEX_CH a simple internal support routine of the Enhancements Module. It examines an input character and determines whether it is a "Ø" through "9" or "A" through "F" (note upper case only). If it is one of these characters, VALID_HEX_CH returns the flag VALIDITY as true; otherwise VALIDITY is false.

5. Routine Relationships Diagram

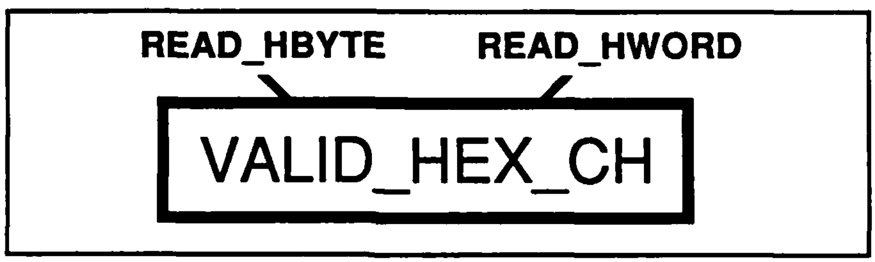


Figure 15. Relationship of VALID_HEX_CH to Other Routines.

6. Invocation

a. Invocation Statement

VALID_HEX_CH is invoked via:

VALIDITY := VALID_HEX_CH(CHARACTER)

where both VALIDITY and CHARACTER are of type Byte.

b. Parameter Passing Schema

VALID_HEX_CH has one input and one output parameter. CHARACTER is passed into the routine and is checked against characters "Ø" through "9" and "A" though "F". VALIDITY is returned as either true if CHARACTER checks out. Otherwise VALIDITY is returned as false.

c. Routines Which Call

VALID_HEX_CH is an internal routine of Enhancements Module. It was written to be called only by other Enhancements Module routines. VALID_HEX_CH is called by READ_HBYTE and READ_HWORD.

7. Variables and Constants

a. Global

VALID_HEX_CH uses no global constants or variables.

b. Module Level

VALID_HEX_CH uses two module constants: TRUE - value 1 hex, logical true; FALSE - value Ø hex, logical false.

c. Routine

VALID_HEX_CH hs no routine level constants or variables.

8. Other Routines Called

VALID_HEX_CH calls no other routines

9. Output of Routine

a. Parameter Passing Schema

VALID_HEX_CH has a single output parameter, VALIDITY, of type Byte. It is returned with the logical value true if the input CHARACTER is a "Ø" through "9" or "A" through "F" . Otherwise VALIDITY is returned with the logical value false.

b. System Configuration Changes

VALID_HEX_CH causes no configuration changes.

10. Routine Testing

a. Description of Test

VALID_HEX_CH was tested in conjunction with the rest of the Enhancements Module rather than being individually tested.

b. Results of Test

VALID_HEX_CH works.

11. Reference to Listing

The listing of VALID_HEX_CH is on page 287 in Appendix A.

1. Routine Names: **WRITE** and **WRITELN**

2. Output routine of Enhancements Module.

3. Written in PLZ.
   WRITE: eight lines of executable code.
   WRITELN: three lines of executable code.

4. Synopsis of Routine

WRITE and WRITELN emulate their Pascal namesakes; they output strings of characters to the device designated by LOGICAL_UNIT. WRITE and WRITELN both use the PLZ STREAM.IO Module routine PUTSEQ to perform the actual output. The difference between the two routines is WRITELN outputs a carrage return at the end of the sequence of characters; WRITE doesn't. WRITELN calls WRITE to output the string and then adds the carriage return via PUTSEQ.

5. Routine Relationships Diagram

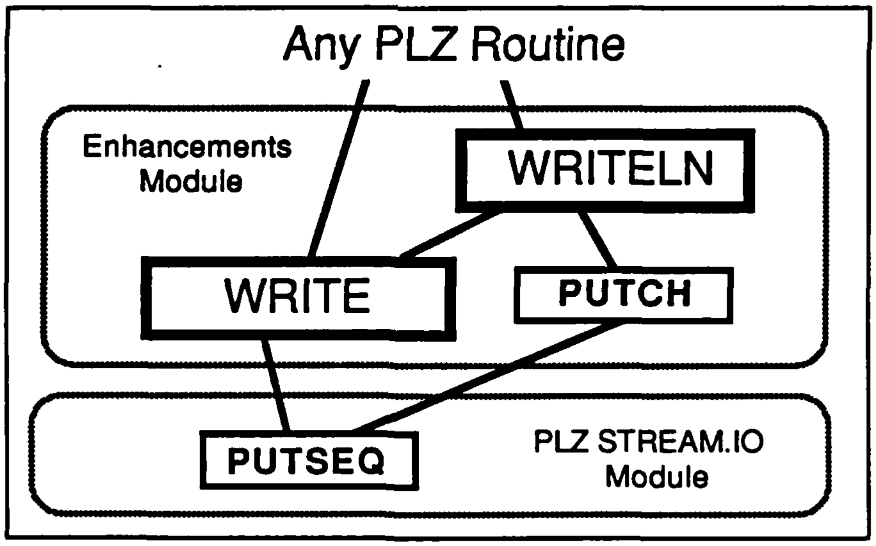


Figure 16. Relationship of WRITE and WRITELN to Calling Routines and PUTSEQ.

6. Invocation

a. Invocation Statement

WRITE and WRITELN are invoked with:

WRITE(LOGICAL_UNIT, BUFFER_PTR) and

WRITELN(LOGICAL_UNIT, BUFFER_PTR)

where LOGICAL_UNIT is type Byte and BUFFER_PTR is of type Pointer-to-Byte.

b. Input Parameter Passing Schema

Both WRITE and WRITELN have two input parameters, LOGICAL_UNIT and BUFFER_PTR. LOGICAL_UNIT brings in the number of the device the output is to go to. BUFFER_PTR points to the memory location where the first character of string to be output is located.

c. Routines Which Call

As global routines of the Ehancements module, WRITE & WRITELN can be called by any PLZ routine which is linked in with Enhancements module. In addition to this purpose, WRITE is used by some other routines in Enhancements Module. Specifically, WRITE is called by WRITE_LBYTE and WRITE_RCODE.

7. Variables and Constants

a. Global

Neither WRITE nor WRITELN use any global variables or constants.

b. Module

Neither routine uses any module level variables. WRITE uses the Enhancements Module constant CARRIAGE_RETURN.

c. Routine

WRITE uses three routine level variables, LENGTH (type Word), RETURN_CODE (type Byte), and PINDEX (type Pointer-To-Byte). LENGTH is used to pass the length of the output character string to the external routine PUTSEQ. RETURN_CODE receives the system completion code sent back from PUTSEQ. PINDEX is a place keeper pointer for the string to be output. WRITELN uses no module level variables. Neither routine uses any routine level constants.

8. Other Routines Called

In addition to WRITELN's calling of WRITE, WRITE calls the external routine PUTSEQ to output strings characters and WRITELN calls PUTCH to output the carriage return.

a. PUTSEQ

This PLZ STREAM.IO Module routine is declaired external to the Enhancements Module. WRITE uses PUTSEQ to output the string of characters to the desinated device driver. PUTSEQ has three input parameters, LOGICAL_UNIT (type Byte), BUFFER_PTR (type Pointer-to-Byte), and LENGTH (type Word), *and has two return parameters*, LENGTH (type Word) and RETURN_CODE (type Byte). LOGICAL_UNIT is the same as the input parameter to WRITE and WRITELN, the number of the device driver to which the output will be directed. BUFFER_PTR points to the first character of the string to be output. LENGTH is the number of characters (Bytes) to be output. The return parameter LENGTH carries the number of characters which were output by PUTSEQ. RETURN_CODE returns the operating system completion code or error code for the output operation. PUTSEQ is invoked via:

```
LENGTH, RETURN_CODE :=
                  PUTSEQ( LOGICAL_UNIT, BUFFER_PTR, LENGTH ).
```

b. PUTCH

PUTCH is an internal support routine of the Enhancements Module. It has two input parameters, LOGICAL_UNIT and CHARACTER, both of type Byte. LOGICAL_ UNIT holds the number of the device driver to which the character is to be output. CHARACTER passes the character to be output. PUTCH is invoked with:

```
PUTCH( LOGICAL_UNIT, CHARACTER ).
```

From WRITELN, CHARACTER passes "%R", the RIO constant for a carriage return. PUTCH has no return parameters.

9. Output of Routine

Neither WRITE or WRITELN have output parameters. Nor does either routine affect the system configuration beyond writing characters to some logical unit.

10. Routine Testing

a. Description of Test

WRITE and WRITELN were tested along with the rest of the Enhancements module routines. A module of test routines called TEST_IT was used to out- put strings to the system console via WRITE and WRITELN.

b. Results of Test

WRITE and WRITELN performed properly.

11. Reference to Listing

The listing of WRITE and WRITELN are on page 288 in Appendix A.

1. Routine Names:
   WRITE_DBYTE, WRITE_HBYTE, WRITE_BBYTE,
   WRITELN_DBYTE, WRITELN_HBYTE, and **WRITELN_BBYTE**

2. Output routines of Enhancements Module.

3. Written in PLZ.

   WRITE_DBYTE: five lines of executable code.
   WRITE_HBYTE: five lines of executable code.
   WRITE_BBYTE: five lines of executable code.
   WRITELN_DBYTE: three lines of executable code.
   WRITELN_HBYTE: three lines of executable code.
   WRITELN_BBYTE: three lines of executable code.

4. Synopsis of Routines

These six routines take a single byte value and output the ASCII characters which represent it. The DBYTE routines output the value in base 1Ø as a decimal value, one to three characters (Ø through 9 or space) followed by a decimal point. The DBYTE routines blank leading zeros in the 1ØØ's and 1Ø's places. The HBYTE routines output the value in hexidecimal form, two characters (Ø to 9 and A to F) followed by an H. The BBYTE routines output a binary representation of the value, eight chararcters (Ø & 1) followed by a B. The WRITE form of the routines does not output a carriage return at the end of the string; the WRITELN forms do. It is up to the calling routine to put CHARACTER in the proper form prior to calling any of the WRITE or WRITELN routines. For example, a number stored in complements form would have to be transformed before WRITE_DBYTE was called. The WRITELN forms function by calling the WRITE version to output the character strings and then call another routine to output the carriage return.

All three WRITE routines function identically; the only difference between them is the values assigned to the internal variables BLANKING and INDEX and the output base value (1Ø, 16, or 2) passed to routine PLACE_LOOP. PLACE_LOOP performs the actual conversion of the byte value into the character string given the base desired and the order or INDEX of the most significant output character. The values for the three routines are:

Routine	BLANKING	INDEX	Base
WRITE_DBYTE	TRUE	1ØØ	1Ø
WRITE_HBYTE	FALSE	16	16
WRITE_BBYTE	FALSE	128	2

5. Routine Relationships Diagram

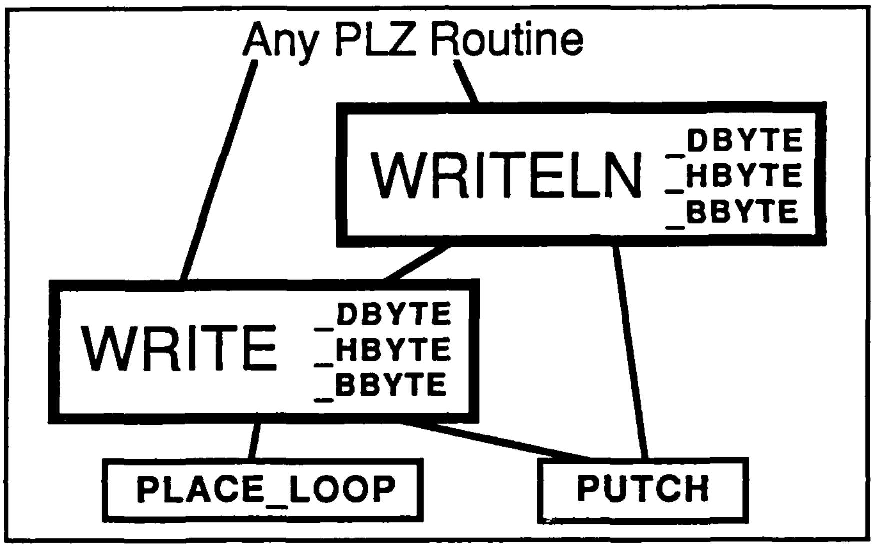


Figure 17. Relationship of Byte WRITE_xBYTE and WRITELN_xBYTE Routines to Other Routines

6. Invocation

a. Invocation Statement

The routines are invoked from calling PLZ routines via:

```
WRITE_DBYTE( LOGICAL_UNIT, NUMBER )
WRITELN_DBYTE( LOGICAL_UNIT, NUMBER )
WRITE_HBYTE( LOGICAL_UNIT, NUMBER )
WRITELN_HBYTE( LOGICAL_UNIT, NUMBER )
WRITE_BBYTE( LOGICAL_UNIT, NUMBER )
WRITELN_BBYTE( LOGICAL_UNIT, NUMBER )
```

where LOGICAL_UNIT and NUMBER are of type Byte.

b. Parameter Passing Schema

All six routines have the same two input parameters, LOGICAL_UNIT and NUMBER, both of type Byte. LOGICAL_UNIT is the number of the device the characters are to be output to. NUMBER is the value to be translated into decimal, hexidecimal, or binary character representations.

c. Routines Which Call

These six routines can be called by any PLZ program. The Enhancements Module and the PLZ Stream.IO Module must be linked in with the calling programs' module.

7. Variables and Constants

a. Global

None of the routines use any global variables or constants aside from the definitions of ASCII characters.

b. Module

None of the routines use any module level variables; The WRITE form routines use no module level constants. The WRITELN forms use the PLZ constant %R to represent a carrage return.

c. Routine

The WRITELN form routines use no routine level constants or variables. The WRITE forms use two variables, BLANKING of type Byte and INDEX of type Word. BLANKING is used as a logical flag to indicate to routine PLACE_LOOP whether leading zeros are to be blanked. INDEX is used to pass the value of the most significant place of the output string to routine PLACE_LOOP. Neither of these variables are necessary, they are present solely to aid the readability of the routines.

8. Other Routines Called

The WRITE and WRITELN routines call two internal routines of the Enhancements Module, PUTCH and PLACE_LOOP.

a. PUTCH

All six routines call PUTCH to output single characters to the desired logical unit. WRITE_DBYTE outputs a decimal point, WRITE_HBYTE outputs an H, WRITE_ BBYTE outputs a B, and the WRITELN's output a carriage return. In all cases PUTCH is invoked via:

PUTCH(LOGICAL_UNIT, CHARACTER)

where both LOGICAL_UNIT and CHARACTER are of type Byte. LOGICAL_UNIT is the same as the input parameter to the WRITE and WRITELN routines, the number of the device to which the CHARACTER is to be written. CHARACTER is the hex value of the ASCII character to be output. PUTCH does not check to see if the CHARACTER is valid ASCII. As the WRITE and WRITELN routines use PUTCH to output constants no error checking is needed. PUTCH has no return parameters.

b. PLACE_LOOP

PLACE_LOOP is called by the three WRITE form routines to translate a value into a string of characters which represent that value and to output those characters to a designated device. PLACE_LOOP is invoked in the three Write routines with:

PLACE_LOOP(LOGICAL_UNIT, BLANKING,
WORD(NUMBER), INDEX, BASE)

where INDEX is of type Word, NUMBER is of type Byte converted to type Word, and the other three input parameters are of type Byte. LOGICAL_UNIT is the same as the input parameter to the WRITE and WRITELN routines, the number of the device to which the string of characters is to be written. BLANKING is a logical flag indicating whether leading zeros are to be blanked. NUMBER is the value to be translated into a string of characters. Note that the input parameter to the WRITE and WRITELN·routines NUMBER is of type Byte and the input to PLACE_LOOP is of type Word. Thus the type conversion in the invocation of PLACE_LOOP. INDEX is the value of the most significant character to be output. BASE is the base in which the character representation is to be made. PLACE_LOOP has no output parameters.

PLACE_LOOP does no range checking on its inputs. This is not a problem as the WRITE routines pass BLANKING, INDEX, and BASE as constants. With the constants passed and the input NUMBER limited to a single byte range, the inputs to PLACE_LOOP cannot be out side defined ranges. It is assumed that the correct LOGICAL_UNIT number is passed into the WRITE and WRITELN routines.

9. Output of Routines

The six routines have no output parameters. The only effect they have on the configuration of the system is the writing of a number characters (two to ten) to some logical unit.

1Ø. Routine Testing

a. Description of Test

These six routines were tested in conjunction with the rest of the Enhancements module routines. Each routine was given a number of values to output.

b. Results of Test

Each routine output its test values in the proper formats.

11. Reference to Listing

The listings for these routines are found on the following pages.

Routine	Page
WRITE_DBYTE	289 in Appendix A
WRITELN_DBYTE	289 in Appendix A
WRITE_HBYTE	290 in Appendix A
WRITELN_HBYTE	290 in Appendix A
WRITE_BBYTE	291 in Appendix A
WRITELN_BBYTE	291 in Appendix A

1. Routine Name: **WRITE_LBYTE** and **WRITELN_LBYTE**

2. Output routines of Enhancements Module.

3. Written in PLZ.
   WRITE_LBYTE: six lines of executable code.
   WRITELN_LBYTE: three lines of executable code.

4. Synopsis of Routines

These two routines take a single byte defined as a logical value and output the text string equivillent of the byte's value. Three string outputs are possible. If the value of the byte is uniary, "TRUE " is output. If the value is zero, "FALSE" is output. If the byte has any other value, the output is "UNDF ". Note that all three output strings are five characters long. The difference between WRITE_LBYTE and WRITELN_BYTE is the same as in Pascal; WRITE_LBYTE does not output a carriage return and WRITELN_ LBYTE does. WRITELN_ LBYTE calls WRITE_LBYTE to perform the five character string output and then calls PUTCH to output the cariage return.

5. Routine Relationships Diagram

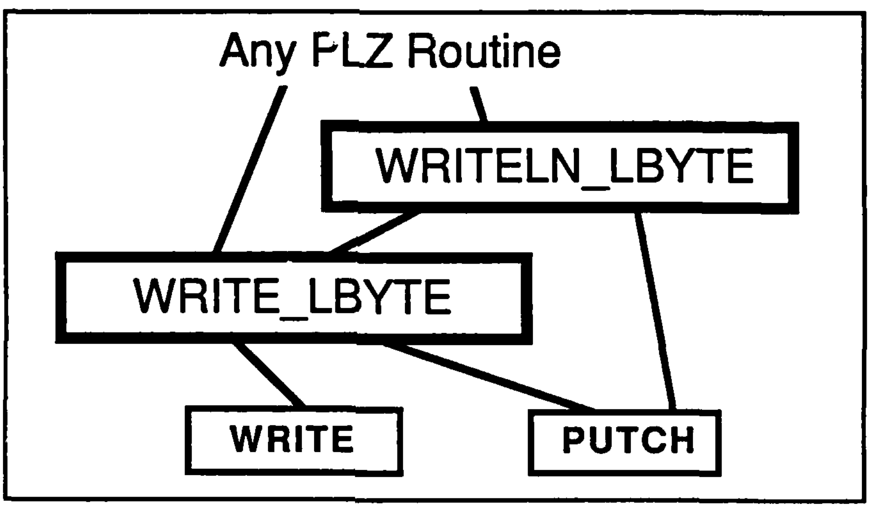


Figure 18. Relationship of Logical-Byte Write and Writeln Routines to Other Routines

6. Invocation

a. Invocation Statement

The routines are invoked from a calling PLZ routine by:

WRITE_LBYTE(LOGICAL_UNIT, FLAG)
WRITELN_LBYTE(LOGICAL_UNIT, FLAG)

where LOGICAL_UNIT and FLAG are both of type Byte.

b. Parameter Passing Schema

Both routines have two input parameters, LOGICAL_UNIT and FLAG. LOGICAL_UNIT is the number of the device the character string is to be written to. FLAG holds the logical variable to be translated into text.

c. Routines Which Call

Both routines can be used by any PLZ language program with which the Enhancements Module and the PLZ Stream.IO Module have been linked. The routines, like the rest of the global Enhancements module routines, are Pascal-like IO subroutines intended to reduce the difficulty of IO in PLZ.

7. Variables and Constants

a. Global

No global variables are used by either routine. Both routines require logical true to be defined as Ø1 hex and logical false to be defined as ØØ hex. Both routines also follow the PLZ convention of "%R" representing a carriage return.

b. Module

Neither routine uses any module level variables. Within the Enhancements Module, TRUE is a constant of value Ø1 hex representing logical true and FALSE is a constant of value ØØ hex representing logical false.

c. Routine

Routine level variables and constants are not used by either routine.

8. Other Routines Called

Both WRITE_LBYTE and WRITELN_LBYTE use other Enhancements Module routines to output characters. WRITE_LBYTE uses the global routine WRITE and WRITELN_LBYTE uses the internal routine PUTCH.

a. WRITE

WRITE is very similar to its Pascal namesake. It outputs a designated string of characters. WRITE_LBYTE uses WRITE to output "TRUE ", "FALSE", or "UNDF " to the designated logical unit. WRITE has two input parameters LOGICAL_UNIT, ot type Byte, and TEXT_POINTER, of type pointer-to-byte or Pbyte. WRITE's LOGICAL_UNIT services the same function as WRITE_LBYTE's input parameter LOGICAL_UNIT. It is the name of the device to which the characters will be written. TEXT_POINTER is a pointer (two bytes) to a specific memory location, the location of the string to be output. For WRITE_ LBYTE, the string is entered as a constant in the invocations of WRITE. PLZ translates this into a pointer to the first character of the string. The "%R" (carriage return) is used by PLZ to denote end-of-string. Thus the invocation of WRITE from WRITE_LBYTE looks like the following.

```
WRITE ( LOGICAL_UNIT, 'string to be output%R' )
```

WRITE has no return parameters.

b. PUTCH

WRITELN_LBYTE uses PUTCH to output a carriage_return to the designated logical unit. PUTCH has two input parameters, LOGICAL_UNIT and CHARACTER. As with WRITE, LOGICAL_UNIT is the Byte parameter indicating which device the output is to go to. CHARACTER, also of type Byte, holds the ASCII character to be output. For WRITELN_LBYTE, PUTCH is invoked by:

```
PUTCH( LOGICAL_UNIT, '%R' )
```

where '%R' denotes a carriage_return. PUTCH has no return parameters.

9. Output of Routine

Neither routine has any output parameters. The sole effect of the routines upon the system is the writing of five characters and, if WRITELN_LBYTE, a carriage return to the designated logical unit.

10. Routine Testing

a. Description of Test

WRITE_LBYTE AND WRITELN_LBYTE were tested along with the rest of Enhancements Module. This test was accomplished by linking with Enhancements Module and the PLZ STREAM.IO Module with a module of test routines.

b. Results of Test

WRITE_LBYTE and WRITE_LBYTE output the correct text strings to the correct logical units.

11. Reference to Listing

The listings of WRITE_LBYTE and WRITELN_LBYTE are on page 292 in Appendix A.

1. Routine Names: **WRITE_DINTEGER** and **WRITELN_DINTEGER**

2. Output routines of Enhancements Module.

3. Written in PLZ. WRITE_DINTEGER: 11 lines of executable code.
WRITELN_DINTEGER: 3 lines of executable code.

4. Synopsis of Routine

WRITE_DINTEGER and WRITELN_DINTEGER take a PLZ Integer type value, translate it into the ASCII characters that represent the base 1Ø magnitude of the value, and then output the characters to a specified logical unit. Since Integer type values have sign, WRITE_DINTEGER and WRITELN_DINTEGER put a blank or a "-" ahead of the character string to indicate the sign of the value. After the last character, the routines output a decimal point. Then, WRITELN_DINTEGER only outputs a carriage_return. Both routines blank leading zeros.

WRITE_DINTEGER does most of the work for both routines as WRITELN_DINTEGER's first statement is a call of WRITE_DINTEGER. WRITE_DINTEGER first determines the sign of the value and outputs a blank for positive or a "-" for negative via routine PUTCH. If the value was negative, it is converted to a positive value, the sign already output. WRITE_DINTEGER then calls PLACE_LOOP to perform the actual translation of value to characters. WRITE_DINTEGER ends by outputting a decimal point via PUTCH. WRITELN_DINTEGER ends by outputting a carriage return.

5. Routine Relationships Diagram

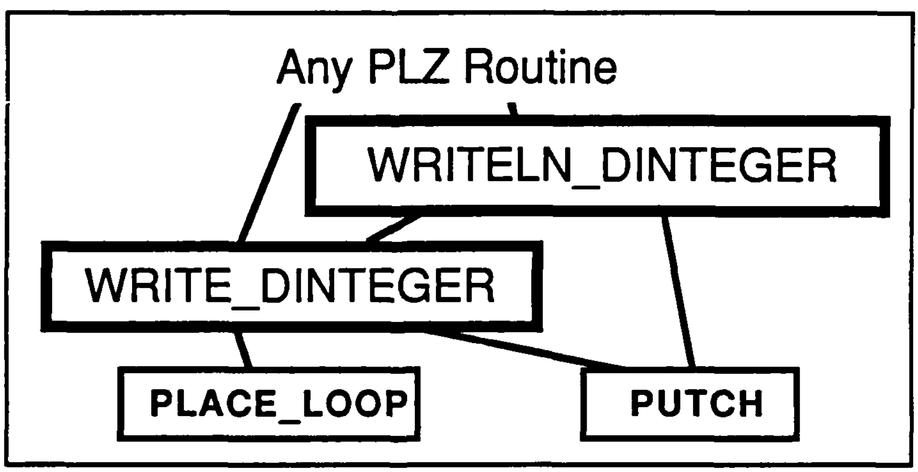


Figure 19. Relationship of WRITE_DINTEGER and WRITELN_DINTEGER to Other Routines.

6. Invocation

a. Invocation Statement

The routines are invoked from calling PLZ programs via:

```
WRITE_DINTEGER( LOGICAL_UNIT, IN_INTEGER )
WRITELN_DINTEGER( LOGICAL_UNIT, IN_INTEGER )
```

where LOGICAL_UNIT is type Byte and IN_INTEGER is type Integer.

b. Parameter Passing Schema

Both routines have two input parameters, LOGICAL_UNIT and IN_INTEGER. LOGICAL_UNIT is the number of the device the output characters are to go to. IN_INTEGER is the value to be output in character form.

c. Routines Which Call

WRITE_DINTEGER and WRITELN_DINTEGER can be called by any PLZ program that has been linked with the Enhancements and PLZ STREAM.IO modules.

7. Variables and Constants

a. Global

Neither routine uses any global variables. WRITE_DINTEGER uses no global constants. WRITELN_DINTEGER follows the PLZ convention of %R representing a carriage return.

b. Module

Neither routine uses any module level variables and WRITELN_DINTEGER uses no module level constants. WRITE_DINTEGER uses the Enhancements Module constant TRUE which represents logical true.

c. Routine

WRITELN_DINTEGER uses no routine level variables or constants.

WRITE_ DINTEGER does have three variables, BLANKING (type Byte), INDEX (tyne Word), and NUMBER (type Word). BLANKING is a logical flag to routine PLACE_ LOOP to indicate whether leading zeros are to be blanked. It is set to TRUE. INDEX passes the place value of the most significant character of the output string to PLACE_ LOOP; INDEX is set to 1ØØØØ decimal. Neither of these variable are necessary though, constants could have been used. These variables are present only to aid routine documentation. NUMBER on the other hand, is used to pass the input Integer value to PLACE_LOOP which uses a Word type input.

8. Other Routines Called

WRITE_DINTEGER and WRITELN_DINTEGER call two internal routines of the Enhnancements module, PUTCH and PLACE_LOOP.

a. PUTCH

Both routines call PUTCH to output single characters. WRITE_ DINTEGER uses PUTCH to output the sign of the value, a blank or a "-", and to output the decimal point. WRITELN_DINTEGER uses PUTCH to putput its carriage return. PUTCH is invoked via:

PUTCH(LOGICAL_UNIT, CHARACTER)

where both input parameter are type Byte. LOGICAL_UNIT is the same as the input parameter parameter LOGICAL_UNIT for WRITE_DINTEGER and WRITELN_DINTEGER. CHARACTER holds the ASCII character to be output. PUTCH has no return parameters.

b. PLACE_LOOP

PLACE_LOOP translates an input value into the characters that represent that value. PLACE_LOOP is called by:

PLACE_LOOP(LOGICAL_UNIT, BLANKING, NUMBER, INDEX, BASE)

where INDEX and NUMBER are type WORD and LOGICAL_UNIT, BLANKING, and BASE are type Byte. LOGICAL_UNIT is the device number for output. BLANKING is a logical flag indicating whether leading zeros are to be blanked. NUMBER is the value to be converted to text representation. INDEX is the place-value of the most significant character to be output. BASE is the base the output string is to be in. PLACE_LOOP has no return parameters.

9. Output of Routine

WRITE_DINTEGER and WRITELN_DINTEGER have no output parameter and only effect the system by outputing seven characters, and a carriage return if WRITELN_ DINTEGER, to some logical unit.

10. Routine Testing

a. Description of Test

WRITE_DINTEGER and WRITELN_DINTEGER were tested along with the other Enhancements Module routines though the module TEST_IT. TEST_IT routines exercised WRITE_DINTEGER and WRITELN_DINTEGER.

b. Results of Test

Both routines performed properly.

11. Reference to Listing

The program listings for WRITE_DINTEGER and WRITELN_ DINTEGER can be found on pages 293 in Appendix A.

1. Routine Names:
   WRITE_DWORD, WRITE_HWORD,
   WRITELN_DWORD, and WRITELN_HWORD

2. Output routines of Enhancements Module.

3. Written in PLZ.

   WRITE_DWORD: five lines of executable code.
   WRITE_HWORD: five lines of executable code.
   WRITELN_DWORD: three lines of executable code.
   WRITELN_HWORD: three lines of executable code.

4. Synopsis of Routines

These four routines take a Word value and output the ASCII characters which represent it. The DWORD routines output the value in base 1Ø as a decimal value, one to five characters (Ø through 9 or space) followed by a decimal point. The DWORD routines blank leading zeros in the 1Ø,ØØØs, 1,ØØØs, 1ØØs, and 1Øs places. The HWORD routines output the value in hexidecimal form, four characters (Ø to 9 and A to F) followed by an H. The WRITE form of the routines does not output a carriage return at the end of the string; the WRITELN forms do. The WRITELN forms function by calling the WRITE version to output the character strings and then call PUTCH to output the carriage return.

Both WRITE routines function identically; the only difference between them is the values assigned to the internal variables BLANKING and INDEX and the output base value (1Ø or 16) passed to routine PLACE_LOOP. PLACE_LOOP performs the actual conversion of the WORD value into the character string given the base desired and the order or INDEX of the most significant output character. The values for the routines are:

Routine	BLANKING	INDEX	Base
WRITE_DWORD	TRUE	1Ø,ØØØ	1Ø
WRITE_HWORD	FALSE	4,Ø96 (1ØØØ hex)	16

5. Routine Relationships Diagram

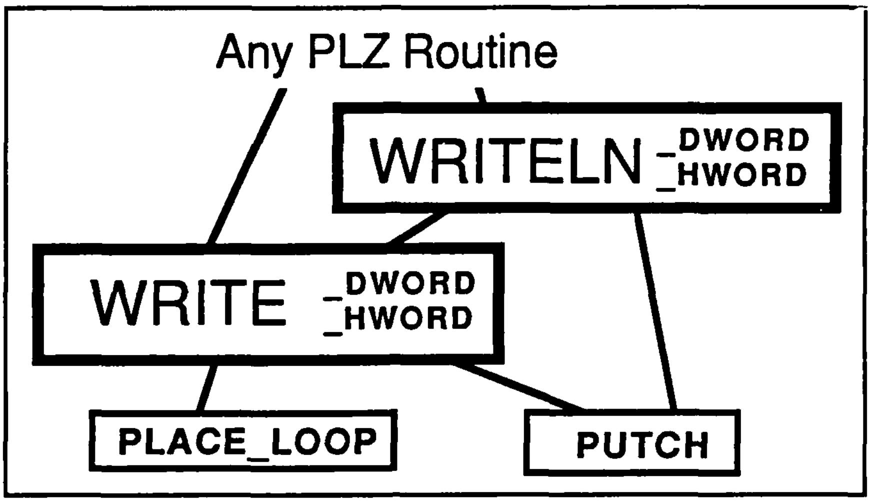


Figure 20. Relationship of Decimal and Hexidecimal Word Write and Writeln Routines to Other Routines.

6. Invocation

a. Invocation Statement

The routines are invoked from calling PLZ routines via:

WRITE_DWORD(LOGICAL_UNIT, NUMBER)
WRITELN_DWORD(LOGICAL_UNIT, NUMBER)
WRITE_HWORD(LOGICAL_UNIT, NUMBER)
WRITELN_HWORD(LOGICAL_UNIT, NUMBER)

where LOGICAL_UNIT is type Byte and NUMBER is type Word.

b. Parameter Passing Schema

All four routines have the same two input parameters, LOGICAL_ UNIT, type Byte, and NUMBER, type Word. LOGICAL_UNIT is the number of the device the characters are to be output to. NUMBER is the value to be translated into decimal or hexidecimal character representations.

c. Routines Which Call

These routines can be called by any PLZ program. The Enhancements Module and the PLZ Stream.IO Module must be linked in with the calling programs' module.

7. Variables and Constants

a. Global

None of the routines use any global variables or constants aside from the definitions of ASCII characters.

b. Module

None of the routines use any module level variables; The WRITE form routines use no module level constants. The WRITELN forms use the PLZ constant %R to represent a carrage return.

c. Routine

The WRITELN form routines use no routine level constants or variables. The WRITE forms use two variables, BLANKING of type WORD and INDEX of type Word. BLANKING is used as a logical flag to indicate to routine PLACE_LOOP whether leading zeros are to be blanked. INDEX is used to pass the value of the most significant place of the output string to routine PLACE_LOOP. These variables could have been constants; they are present solely to aid the readability of the routines.

8. Other Routines Called

The WRITE and WRITELN routines call two internal routines of the Enhancements Module, PUTCH and PLACE_LOOP.

a. PUTCH

All four routines call PUTCH to output single characters to the desired logical unit. WRITE_DWORD outputs a decimal point, WRITE_HWORD outputs an H, and the WRITELN's output a carriage return. In all cases PUTCH is invoked via:

PUTCH(LOGICAL_UNIT, CHARACTER)

where both LOGICAL_UNIT and CHARACTER are type Byte. LOGICAL_UNIT is the same as the input parameter to the WRITE and WRITELN routines, the number of the device to which the CHARACTER is to be written. CHARACTER is the hex value of the ASCII character to be output. PUTCH does not check to see if the CHARACTER is valid ASCII. As the WRITE and WRITELN routines use PUTCH to output constants no error checking is needed. PUTCH has no return parameters.

b. PLACE_LOOP

PLACE_LOOP is called by the WRITE form routines to translate a value into a string of characters which represent that value and to output those characters to a designated device. PLACE_LOOP is invoked by:

PLACE_LOOP(LOGICAL_UNIT, BLANKING, NUMBER , INDEX, BASE)

where INDEX is of type Word, NUMBER is of type WORD converted to type Word, and the other three input parameters are of type WORD. LOGICAL_UNIT is the same as the input parameter to the WRITE and WRITELN routines, the number of the device to which the string of characters is to be written. BLANKING is a logical flag indicating whether leading zeros are to be blanked. NUMBER is the value to be translated into a string of characters. Note that the input parameter to the WRITE and WRITELN routines NUMBER is of type WORD and the input to PLACE_LOOP is of type Word. Thus the type conversion in the invocation of PLACE_LOOP. INDEX is the value of the most significant character to be output. BASE is the base in which the character representation is to be made. PLACE_LOOP has no output parameters.

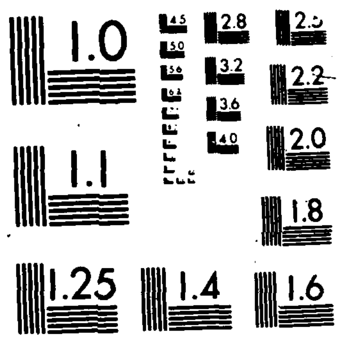
PLACE_LOOP does no range checking on its inputs. This is not a problem as the WRITE routines pass BLANKING, INDEX, and BASE as constants. With the constants passed and the input NUMBER limited to a single WORD range, the inputs to PLACE_LOOP cannot be out side defined ranges. It is assumed that the correct LOGICAL_UNIT number is passed into the WRITE and WRITELN routines.

9. Output of Routines

The routines have no output parameters. The only effect they have on the configuration of the system is the writing of a number characters (six to seven) to some logical unit.

AD-A172 823 DESIGN AND PARTIAL IMPLEMENTATION OF A COMPUTER CONTROLLED DATA COLLECTION SYSTEM(U) AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH SCHOOL OF ENGI.. L E LUTZ 2/5

UNCLASSIFIED FEB 86 AFIT/GE/ENG/86M-1 F/G 9/2 NL


1.0
2.8
2.5
3.2
2.2
3.6
4.0
2.0
1.1
1.8
1.25
1.4
1.6

1Ø. Routine Testing

a. Description of Test

These routines were tested in conjunction with the rest of the Enhance ments Module routines via the TEST_IT Module. Each Enhancements Module routines was given a number of values to output.

b. Results of Test

Each routine output its input values in the proper formats.

11. Reference to Listing

The listings of WRITE_DWORD, WRITELN_DWORD, WRITE_HWORD, and WRITELN_HWORD are on pages 294 in Appendix A and 295 in Appendix A.

1. Routine Name: **WRITE_POINTER** and **WRITELN_POINTER**

2. Output routines of Enhancements Module.

3. Written in PLZ.
   WRITE_POINTER: five lines of executable code.
   WRITELN_POINTER: three lines of executable code.

4. Synopsis of Routines

These two routines take a memory address and output its text string equivalent. The output text string consists of a "^" followed by four hexidecimal characters (Ø to 9 and A to F). WRITE_POINTER does not output a carriage return and WRITELN_ POINTER does. WRITELN_POINTER calls WRITE_ POINTER to perform the character string output and then calls PUTCH to output the cariage return.

5. Routine Relationships Diagram

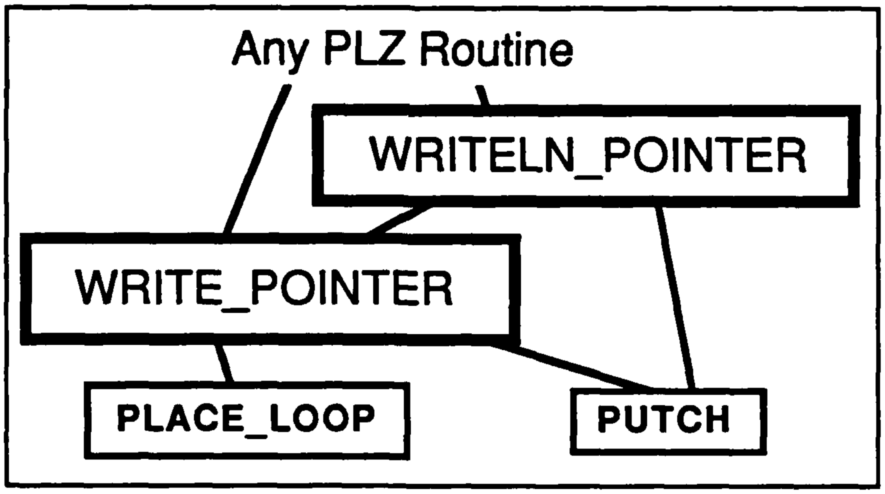


Figure 21. Relationship of Pointer Write and Writeln Routines to Other Routines.

6. Invocation

a. Invocation Statement

The routines are invoked from a calling PLZ routine by:

WRITE_POINTER(LOGICAL_UNIT, LOCATION)
WRITELN_POINTER(LOGICAL_UNIT, LOCATION)

where LOGICAL_UNIT is of type Byte and LOCATION is of type Word.

b. Parameter Passing Schema

Both routines have two input parameters, LOGICAL_UNIT and LOCATION. LOGICAL_UNIT is the number of the device the character string is to be written to. LOCATION holds the address to be translated into text.

c. Routines Which Call

Both routines can be used by any PLZ language program with which the Enhancements Module and the PLZ Stream.IO module have been linked. The routines, like the rest of the global Enhancements module routines, are Pascal-like IO subroutines intended to reduce the difficulty of IO in PLZ.

7. Variables and Constants

a. Global

No global variables are used by either routine. WRITE_POINTER users no global constants. WRITELN_POINTER follows the PLZ convention of "%R" representing a carriage return.

b. Module

Neither routine uses any module level variables. Within the Enhancements Module, TRUE is a constant of value Ø1 hex representing logical true and FALSE is a constant of value ØØ hex representing logical false.

c. Routine

Routine level constants are not used by either routine. WRITELN_ POINTER has no routine level variables. WRITE_POINTER has two routine level variables, BLANKING (type Byte) and INDEX (type Word). BLANKING is a logical flag to routine PLACE_LOOP to indicate whether leading zeros are to be blanked. INDEX passes to PLACE_LOOP the place-value of the most significant character of the output text string. These two variables exist only to aid the readability of the code.

8. Other Routines Called

Both WRITE_POINTER and WRITELN_POINTER use other Enhancements Module routines to output characters. WRITE_POINTER uses the internal routine PLACE_LOOP to perform the actual value to character string conversion. Both routines use the internal routine PUTCH to output single characters.

a. PLACE_LOOP

PLACE_LOOP translates a value into a string of characters that represents the value and then outputs the characters to a designated logical unit. PLACE_LOOP is invoked with:

PLACE_LOOP(LOGICAL_UNIT, BLANKING, NUMBER, INDEX, BASE)

where NUMBER and INDEX are type Word and LOGICAL_UNIT, BLANKING, and BASE are type Byte. The parameter LOGICAL_UNIT for PLACE_LOOP is the same as the LOGICAL_UNIT input to WRITE_POINTER and WRITELN_ POINTER. It is the number of the device to which the output will go. BLANKING is a logical flag indicating whether leaing zeros are to be blanked. NUMBER is the value to be translated into a string of ASCII characters. INDEX hold the place-value of the most significant character of the output string. BASE is the desired base of the character representation. PLACE_LOOP has no return parameters.

b. PUTCH

PUTCH outputs single characters to the designated logical unit. PUTCH has two input parameters, LOGICAL_UNIT and CHARACTER. As with WRITE, LOGICAL_ UNIT is the Byte parameter indicating which device the output is to go to. CHARACTER, also of type Byte, holds the ASCII character to be output. PUTCH is invoked by:

PUTCH(LOGICAL_UNIT, CHARACTER)

WRITE-POINTER uses PUTCH to output the "^" and WRITELN_POIINTER uses PUTCH to output its carriage return. PUTCH has no return parameters.

9. Output of Routine

Neither routine has any output parameters. The sole effect of the routines upon the system is the writing of five characters and, if WRITELN_ POINTER, a carriage return to the designated logical unit.

10. Routine Testing

a. Description of Test

WRITE_POINTER AND WRITELN_POINTER were tested along with the rest of Enhancements module. This test was accomplished by linking with Enhancements module and the PLZ STREAM.IO module a module of test routines.

b. Results of Test

WRITE_POINTER and WRITELN_POINTER output the correct text strings to the correct logical units.

11. Reference to Listing

The listings of WRITE_POINTER and WRITELN_POINTER are on page 296 in Appendix A.

1. Routine Names: **WRITE_RCODE** and **WRITELN_RCODE**

2. Output routines of Enhancements Module.

3. Written in PLZ.
   WRITE_RCODE: 45 lines of executable code.
   WRITELN_RCODE: 3 lines of executable code.

4. Synopsis of Routines

WRITE_RCODE and WRITELN_RCODE are PLZ routines which translate the RIO (operating system) hexadecimal error codes into their text definitions and outputs the text to the system console. WRITE_RCODE is just one big case statement with 43 cases, one case for each RIO return code. WRITE_RCODE is intended to be linked in during program checkout for rapid diagnosis of operating system problems. WRITE_RCODE does not send a carriage return to the console. In contrast, WRITELN_RCODE consists of two subroutine calls and does send a carrage return to the console at the end of the text string.

5. Routine Relationships Diagram

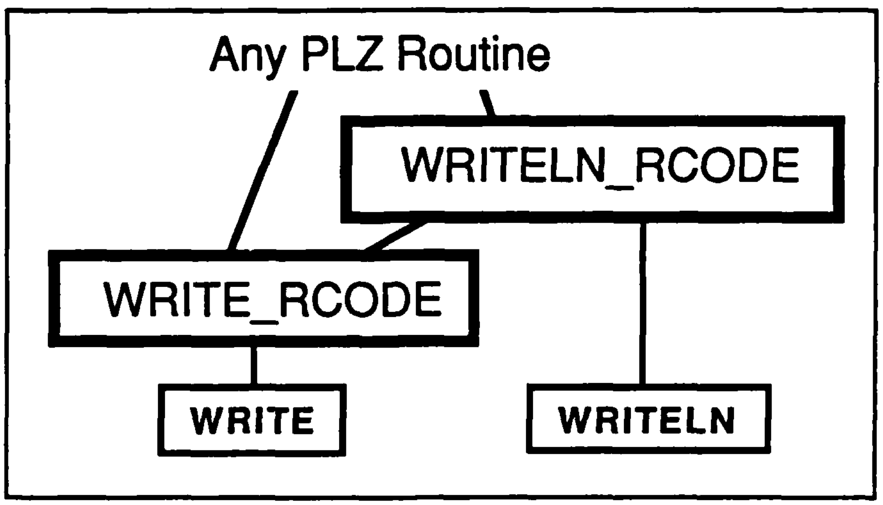


Figure 22. Relationship of WRITE_RCODE and WRITELN_RCODE to Other Routines

6. Invocation

a. Invocation Statement

WRITE_RCODE and WRITELN_RCODE are invoked from another PLZ routine via:

WRITE_RCODE (RETURN_CODE) and

WRITELN_RCODE(RETURN_CODE)

where RETURN_CODE is the RIO code in question. WRITE_RCODE and WRITELN_RCODE must either be linked in and declaired as an external procedure or be compiled with the calling PLZ routine.

b. Parameter Passing Schema

WRITE_RCODE and WRITELN_RCODE both have one input parameter, RETURN_CODE, of type Byte. If either routine is passed an undefined RETURN_CODE, the routine executes without taking any action. See the routine listing for the defined return codes and their text definitions.

c. Routines Which Call WRITE_RCODE and WRITELN_RCODE

WRITE_RCODE and WRITELN_RCODE can be called by any PLZ program they are linked with or compiled with. WRITE_RCODE is called by WRITELN_RCODE to translate the RETURN_CODE into text.

7. Variables and Constants

a. Global

Both routines use the constant "%R", the PLZ representation for a carriage return. %R indicates to routines WRITE and WRITELN the end of the string to be output. Neither routine uses any global variables.

b. Internal to the Module

CONSOLE_OUT, a constant of value two, is used by both WRITE_RCODE and WRITELN_RCODE. It is the logical unit number for the system console. Neither routine uses any module level variables.

c. Internal to the Routine

Neither WRITE_RCODE nor WRITELN_RCODE use any routine level variables or constants.

8. Other Routines Called

WRITE_RCODE calls the routine WRITE and WRITELN_RCODE calls WRITELN to output the text translation of the return codes to the system console. WRITE and WRITELN are also part of Enhancements Module.

WRITE and WRITELN have two input parameters, LOGICAL_UNIT, type byte, and TEXT_POINTER, type PByte for pointer to byte. For both routines, LOGICAL_UNIT is always CONSOLE_OUT or 2. TEXT_POINTER points to the first character of the text string listed in each case of WRITE_RCODE and the carriage return, %R, for WRITELN_RCODE. WRITE and WRITELN are invoked via:

WRITE (LOGICAL_UNIT, #'text string %R') and

WRITELN(LOGICAL_UNIT, #text string %R')

where %R indicates the end of the string and the # is the PLZ indicator for a pointer to a string delimited by single quotes.

9. Output of Routine

WRITE_RCODE and WRITELN_RCODE have no output parameters as such though it does output text to the system console. Neither routine alters the configuration of the system.

10. Routine Testing

No specific tests were created for WRITE_RCODE and WRITELN_ RCODE. Rather, they were used as designed, linked in with other PLZ programs for diagnosis. When errors occured and RIO codes were received, the routiness translated the codes and output the text to the system console. Not only did both routines work, they proved to be a valuable debugging aids.

11. Reference to Listing

The listing of WRITE_RCODE can be found on pages 297-298 in Appendix A. WRITELN_ RCODE's listing is on page 298 in Appendix A.

1. Routine Name: **READLN**

2. Output routine of Enhancements Module.

3. Written in PLZ; six lines of executable code.

4. Synopsis of Routine

READLN is input Enhancement Module routine for the PLZ language; its purpose is input of text strings. READLN reads in ASCII character and places them in a buffer. This continues until a carriage return is read. At that point, READLN returns to the calling routine a pointer to the last character in the buffer, the carriage return.

This READLN is ment to approximate the function of the Pascal Readln command. Unlike the Pascal command, this READLN has two input parameters to indicate from which logical unit the text is to be read from and to provide a pointer to the memory location where the string will be put. To let the calling routine know how long a text string was read in, this READLN returns a pointer to the end of the string. Again, unlike Pascal, the calling routine must ensure sufficent buffer space to accomodate the input string. By using this READLN, a PLZ program can read in text string far more easily than would be possible with the GETSEQ routine of the PLZ STREAM.IO module, though still not as easy as with the Pascal Readln.

5. Routine Relationships Diagram

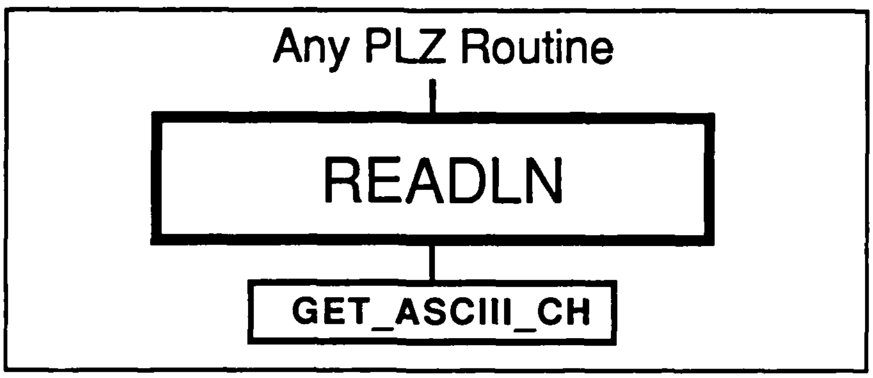


Figure 23. Relationship of READLN to Calling PLZ Routines and to GET_ASCII_CH.

6. Invocation

a. Invocation Statement

READLN is invoked from a calling PLZ routine with:

OUT_POINTER := READLN(LOGICAL_UNIT, TEXT_POINTER)

where all OUT_POINTER and TEXT_POINTER are of type PByte (for pointer to byte) and LOGICAL_UNIT is of type Byte. The calling program must ensure the buffer pointed to by TEXT_PONTER is large enough to accomodate the input text string.

b. Parameter Passing Schema

READLN has two input parameters, LOGICAL_UNIT and TEXT_ POINTER. LOGICAL_UNIT passes the number of the device the text is to read in from. TEXT_ POINTER holds the beginning address of the buffer into which the input text will be copied.

c. Routines Which Call READLN

READLN can be used by any PLZ program linked with the Enhancements Module and the PLZ STREAM.IO Module. Alternately, READLN, along with the internal routines GET_ASCII_CH and GETCH, could be part of the calling program's module. PLZ STREAM.IO will still have to be linked in.

7. Variables and Constants

a. Global

READLN uses no declaired global constants or variables. However, the buffer into which the text string is in a sense a global buffer.

b. *Module*

No module level variables are used by READLN. The module constant CARRIAGE_RETURN, valued at ØD hex, is used by READLN.

c. Routine

READLN has no constants; it uses one variable, PINDEX, of type PByte for Pointer-to-Byte. PINDEX is used as a place keeper, pointing to the current position in the buffer.

8. Other Routines Called

READLN uses GET_ASCII_CH, and internal routine of Enhancements mod- ule, to read in each character. GET_ASCII_CH is invoked from GET_ ASCII_CH by:

PINDEX^ := GET_ASCII_CH(LOGICAL_UNIT)

where PINDEX^ is the byte pointed to by PINDEX (and thus is of type Byte) and LOGICAL_UNIT is of type Byte. PINDEX^ is the memory location into which the charac- ter reterived by GET_ASCII_CH is placed. LOGICAL_UNIT is the device number the character is read from.

9. Output of Routine

READLN returns a single parameter to the calling routine, OUT_ POINTER, of type PByte. OUT_POINTER points to the last character placed in the buffer, the carriage return. Thus, having passed to READLN TEXT_POINTER, pointing to be beginning of the buffer, and having received back OUT_POINTER, the calling routine can determine the length of the string in the buffer. READLN does not alter the configuration of the system beyond changing a number of memory locations to the values read in from the logical unit.

10. Routine Testing

a. Description of Test

READLN and the rest of the read routines of the Enhancements Module were tested with a special module of test routines. One of these routines used READLN to get text in from the keyboard and then displayed it to the system console.

b. Results of Test

READLN performed properly.

11. Reference to Listing

READLN's program listing is on page 299 in Appendix A.

1. Routine Name: **READ_HBYTE**

2. Output routine of Enhancements Module.

3. Written in PLZ; seven lines of executable code.

4. Synopsis of Routine

READ_HBYTE reads in from a designated logical unit two characters representing a 8 bit value in hexidecimal form. The routine translates the characters into the value and returns that value to the calling routine. In reading in the character, READ_HBYTE accepts only valid hexidecimal characters (Ø - 9 and A - F) rejecting all other characters. READ_HBYTE will keep reading in characters until it has read two valid hexidecimal characters.

5. Routine Relationships Diagram

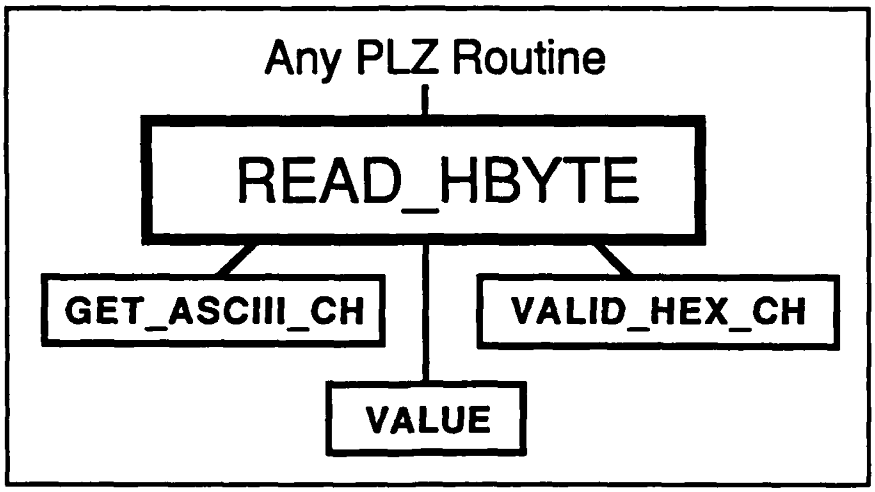


Figure 24. Relationship of READ_HBYTE to Other Routines.

6. Invocation

a. Invocation Statement

READ_HBYTE is invoked from a calling PLZ language routine by:

NUMBER := READ_HBYTE(LOGICAL_UNIT)

where both NUMBER and LOGICAL_UNIT are of type Byte.

b. Parameter Passing Schema

READ_HBYTE has a single input parameter, LOGICAL_UNIT, which holds the number of the device from which the hexidecimal characters are to be read.

c. Routines Which Call

READ_HBYTE can be called by any PLZ routine that is linked with the En- hancements Module and the PLZ STREAM.IO module. Alternately, this routine (and the internal routines GET_ASCII_CH, GETCH, and VALID_HEX_CH) could be part of the calling routine's module. The STREAM.IO module is still required.

7. Variables and Constants

a. Global

No global level variables or constants are used by READ_HBYTE.

b. Module

READ_HBYTE uses no module level variables or constants.

c. Routine

Two local variables are used by READ_HBYTE. FIRST_TERM is the first valid hexidecimal character read in and SECOND_TERM is the second. Both of these variables are of type Byte. READ_HBYTE uses no routine level constants.

8. Other Routines Called

READ_HBYTE calls three internal routines of the Enhancements Module, GET_ASCII_CH, VALID_HEX_CH, and VALUE.

a. GET_ASCII_CH

This routine reads individual ASCII characters in from a designated logical unit. It is invoked by:

CHARACTER := GET_ASCII_CH(LOGICAL_UNIT)

where both CHARACTER and LOGICAL_UNIT are of type Byte. READ_HBYTE uses GET_ASCII_CH to get FIRST_TERM and SECOND_TERM.

b. VALID_HEX_CH

This function determines whether its input character is a valid hexidecimal character (Ø - 9 or A - F). If it is valid, a logical TRUE is returned, otherwise a FALSE is returned. VALID_HEX_CH is invoked with:

VALID_HEX_CH(CHARACTER)

where CHARACTER is of type Byte and VALID_HEX_CH returns as a logical Byte.

c. VALUE

This internal function of the Enhancements module translates a decimal or hexidecimal ASCII character (Ø - 9 and A - F) into the value it represents and returns that value. If VALUE receives an invalid character, a value of zero is returned. The function VALUE is invoked by its name as follows.

VALUE(CHARACTER)

Both CHARACTER and the return VALUE are of type Byte.

9. Output of Routine

READ_HBYTE has a single return parameter and produces no changes in the system configruation. The return parameter, NUMBER, is the hexidecimal (8 bit) value derrived from the two characters read in.

10. Routine Testing

a. Description of Test

READ_HBYTE was tested along with the rest of the Enhancements Module routines. In this test READ_HBYTE read in some values from the keyboard; the values were then output to the screen.

b. Results of Test

READ_HBYTE properly input hexidecimal values and performed as expected for all input data.

11. Reference to Listing

The listing for READ_HBYTE is on page 300 in Appendix A.

1. Routine Name: **READ_DBYTE**

2. Part of Enhancements Module

3. Written in PLZ; ten lines of executable code.

4. Synopsis of Routine

READ_DBYTE reads three characters in from a specified logical unit and translates these characters into the decimal value represented by the characters. The internal Enhancements Module routine GET_ASCII_CH is used for the character input. The first character read in must be valid decimal character, that is Ø through 9. For the first character, all nonvalid decimal characters will be rejected. Character validity is checked by the internal Enhancements Module routine VALID_DECIMAL_CH. If the second or third character read in are invalid they will be accepted but will not be included in the value calculation. The actual conversion of character to value is accom- plished by VALUE, an internal routine of the Enhancements Module.

As a single byte has a maximum value of 255, if the decimal characters represent a value greater than this overflow will occur. The calling routine must guard against this condition as READ_DBYTE does no range checking.

5. Routine Relationships Diagram

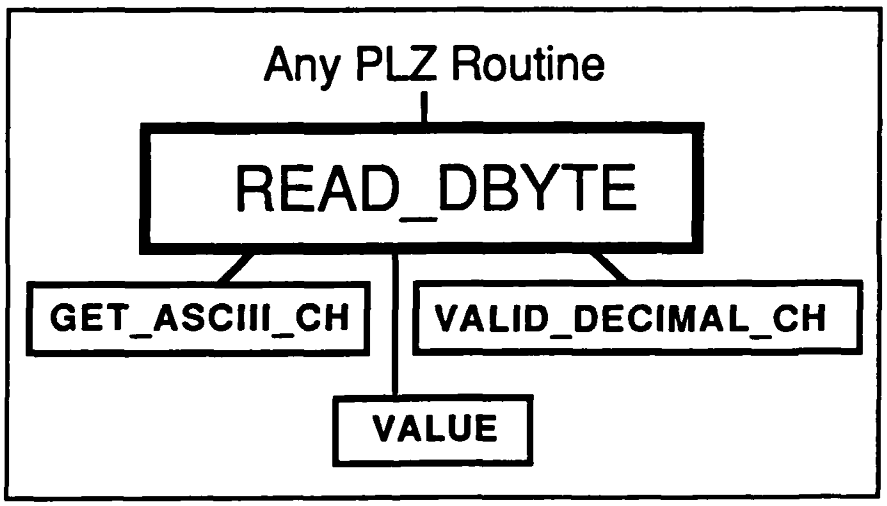


Figure 25. Relationship of READ_DBYTE to Other Routines.

6. Invocation

a. Invocation Statement

READ_DBYTE is invoked through the following statement.

NUMBER := READ_DBYTE(LOGICAL_UNIT)

NUMBER and LOGICAL_UNIT are both of type Byte.

b. Parameter Passing Schema

READ_DBYTE has a single input parameter, LOGICAL_UNIT, the number of the device from which the characters will be read.

c. Routines Which Call

READ_DBYTE can be called by any PLZ routine.

7. Variables and Constants

a. Global

READ_DBYTE uses no global variables or constants.

b. Module

READ_DBYTE uses no module level variables. The module constant TRUE for logical true is used.

c. Routine

READ_DBYTE uses three internal variables, FIRST_TERM, SECOND_TERM, and THIRD_TERM, all of type Byte. These three variables are used to hold the validated characters prior to calculating the decimal value they represent. READ_DBYTE uses no routine level constants.

8. Other Routines Called

READ_DBYTE uses three internal routines of the Enhancements module, GET_ASCII_CH, VALID_DECIMAL_CH, and VALUE.

a. GET_ASCII_CH

This routine reads single characters in from a specified logical unit. GET_ASCII_CH returns only valid ASCII characters. The routine is invoked by:

CHARACTER := GET_ASCII_CH(LOGICAL_UNIT)

where both CHARACTER, and LOGICAL_UNIT are of type Byte.

b. VALID_DECIMAL_CH

This function checks a character to determine whether it is a Ø through 9. If the input character is a valid decimal character, VALID_DECIMAL_CH returns a value of TRUE. Otherwise, a value of FALSE is returned. VALID_DECIMAL_CH is invoked with:

VALID_DECIMAL_CH(CHARACTER)

where CHARACTER and the return VALID_DECIMAL_CH are type Byte.

c. VALUE

This internal function of the Enhancements module translates a decimal or hexidecimal ASCII character (Ø - 9 and A - F) into the value it represents and returns that value. If VALUE receives an invalid character, a value of zero is returned. The function VALUE is invoked by its name as follows.

VALUE(CHARACTER)

Both CHARACTER and the return VALUE are of type Byte.

9. Output of Routine

READ_DBYTE returns to its calling routine a single parameter, NUMBER, which holds value translated from the characters. NUMBER is of type Byte. Other than reading in a number of characters, READ_DBYTE causes no system configuration changes.

10. Routine Testing

a. Description of Test

READ_DBYTE was tested with the rest of the Enhancements module routines via a version of the test module TEST_IT. In this test values were output though READ_DBYTE to the system console.

b. Results of Test

READ_DBYTE performed properly.

11. Reference to Listing

The program listing of READ_DBYTE can be found on page 301 in Appendix A.

1. Routine Name: **READ_BBYTE**

2. Output routine of Enhancements Module.

3. Written in PLZ; thirteen lines of executable code.

4. Synopsis of Routine

READ_BBYTE reads in from a designated logical unit one to eight characters representing a 8 bit value in binary form. The routine translates the characters into the value and returns that value to the calling routine. In reading in the first character, READ_BBYTE accepts only valid binary characters (Ø and 1) rejecting all other characters. READ_BBYTE will keep reading in characters until it has a 1 or Ø. Subsequent 1s and Øs will be read in and included for the value calculation. However, as soon as a character other than a 1 or Ø is read, character input ceases. The character reading is accomplished through routine GET_ASCII_CH.

READ_BBYTE stores the 1s and Øs in a text string which it passes to routine VALUE_LOOP for translation into a value. READ_BBYTE then returns this value to its calling routine.

5. Routine Relationships Diagram

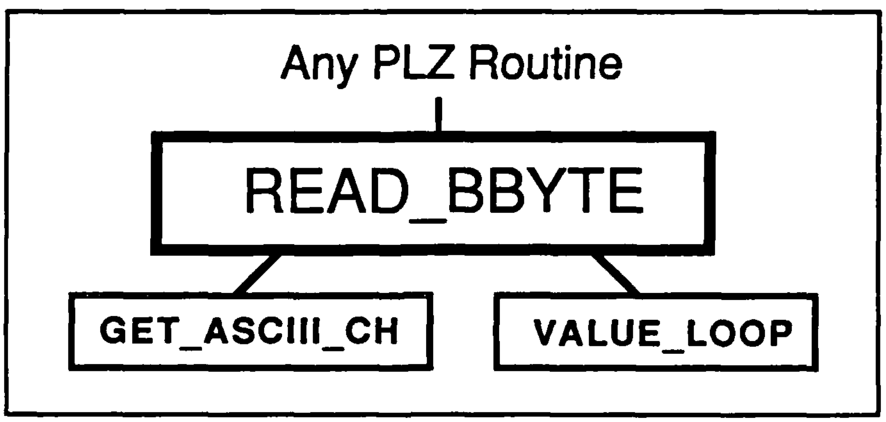


Figure 26. Relationship of READ_BBYTE to Calling PLZ Routine, GET_ASCII_CH, and VALUE_LOOP.

6. Invocation

a. Invocation Statement

READ_BBYTE is invoked from a calling PLZ language routine by:

NUMBER := READ_BBYTE(LOGICAL_UNIT)

where both NUMBER and LOGICAL_UNIT are of type Byte.

b. Parameter Passing Schema

READ_BBYTE has a single input parameter, LOGICAL_UNIT, which holds the number of the device from which the hexidecimal characters are to be read.

c. Routines Which Call

READ_BBYTE can be called by any PLZ routine that is linked with the Enhancements Module and the PLZ STREAM.IO Module. Alternately, this routine (and the internal routines GET_ASCII_CH, GETCH, and VALID_HEX_CH) could be part of the calling routine's module. The STREAM.IO Module is still required.

7. Variables and Constants

a. Global

No global level variables or constants are used by READ_BBYTE.

b. Module

READ_BBYTE uses no module level variables. The Enhancements Module constant BLANK (ASCII for 2Ø hex) is used by READ_BBYTE.

c. Routine

Three local variables are used by READ_BBYTE. INPUT_STRING, of type ASCII_STR, an array of eight Bytes, is used to store the 1s and Øs. INDEX, of type Byte, is a placekeeper for the array INPUT_STRING. The third variable, CHARACTER, is used to hold each character as it is read in.

8. Other Routines Called

READ_BBYTE calls two internal routines of the Enhancements Module, GET_ASCII_CH and VALUE_LOOP.

a. GET_ASCII_CH

This routine reads individual ASCII characters in from a designated logical unit. It is invoked by:

CHARACTER := GET_ASCII_CH(LOGICAL_UNIT)

where both CHARACTER and LOGICAL_UNIT are of type Byte. READ_BBYTE uses GET_ASCII_CH to read in the characters.

b. VALUE_LOOP

This routine translates a string of ASCII characters into the value they repre- sent. VALUE_LOOP is invoked though:

MAGNITUDE := VALUE_LOOP(INPUT_STRING, MULTIPLIER).

VALUE_LOOP has two input parameters, INPUT_STRING (type PByte), a pointer to the string of ASCII characters, and MULTIPLIER (type Word) the base of the number represented by the string of characters. Starting from the least significant character VALUE_ LOOP calculates the value contributed by each character to the total MAGNITUDE represented by the string. The routine ends when a blank is found in the INPUT_STRING or when eight characters have been translated. VALUE_LOOP has a single return parameter, MAGNITUDE, of type Word.

9. Output of Routine

READ_BBYTE has a single return parameter and produces no changes in the system configuration. The return parameter, NUMBER, is the hexidecimal (8 bit) value derrived from the 1s and Øs read in.

10. Routine Testing

a. Description of Test

READ_BBYTE was tested along with the rest of the Enhancements Module routines. In this test READ_BBYTE read in some values from the keyboard; the values were then output to the screen.

b. Results of Test

READ_BBYTE properly read in binary values and converted them to the proper values.

11. Reference to Listing

The listing of READ_BBYTE is on page 302 in Appendix A.

1. Routine Name: **READ_LBYTE**

2. Output routine of Enhancements Module

3. Written in PLZ; seven lines of executable code.

4. Synopsis of Routine

This simple routine reads in characters, on at a time, from a designated logical unit. It the character is a T, t, or 1, a value of logical true is returned to the calling routine. If the character is a F, f, or Ø a value of logical false is returned to the calling routine. If any other character input, the routine loops and another character is read in. READ_LBYTE uses the internal Enhancements module routine GET_ASCII_CH to read in the character(s).

5. Routine Relationships Diagram

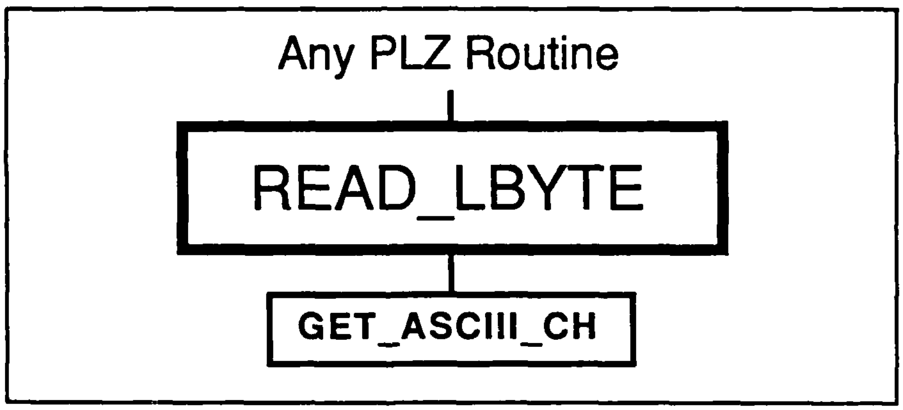


Figure 27. Relationship of READ_LBYTE to Calling Routines and to GET_ASCII_CH.

6. Invocation

a. Invocation Statement

READ_LBYTE is invoked by the following statement.

TRUTH := READ_LBYTE(LOGICAL_UNIT)

Both the input and return parameters are of type Byte.

b. Parameter Passing Schema

READ_LBYTE has one input parameter, LOGICAL_UNIT, the device number from which the character will be read.

c. Routines Which Call

READ_LBYTE, like the rest of the global routines of the Enhancements Module, are ment to be called from any PLZ routine that needs IO assistance.

7. Variables and Constants

a. Global

READ_LBYTE uses no global constants or variables.

b. Module

READ_LBYTE uses the module constants TRUE and FALSE for logical true and false. The routine uses no module level variables.

c. Routine

READ_LBYTE employes the local variable CHARACTER, of type Byte, to hold the character read in. The routine has no locally defined constants.

8. Other Routines Called

READ_LBYTE calls GET_ASCII_CH, an internal routine of the Enhancements module, to read in the character input. GET_ASCII_CH is invoked by:

CHARACTER := GET_ASCII_CH(LOGICAL_UNIT)

where both the input parameter LOGICAL_UNIT and the return character CHARACTER are of type Byte. LOGICAL_UNIT holds the input device number. CHARACTER holds the character input. GET_ASCII_CH returns only valid ASCII characters.

9. Output of Routine

a. Parameter Passing Schema

READ_LBYTE has a single return parameter, TRUTH, of type Byte. TRUTH returns the logical value derived from the read character. TRUTH can take on only the values TRUE or FALSE.

b. System Configuration Changes

The routine causes no system configuration changes aside from reading in a character.

10. Routine Testing

a. Description of Test

READ_LBYTE was tested in the same fashion as the rest of the Enhancements Module routines.

b. Results of Test

The routine performed properly.

11. Reference to Listing

The program listing of READ_LBYTE is on page 303 in Appendix A.

1. Routine Name: **READ_DINTEGER**

2. Output routine of Enhancements Module.

3. Written in PLZ; 22 lines of executable code.

4. Synopsis of Routine

READ_DINTEGER reads in a string of characters from the designated logical unit and translates that string into a signed value. The routine begins by calling GET_ASCII_CH to read in the sign character. Characters are read in from the desired logical unit until a blank, "+", or "-" is read in, these three being the valid sign characters. The sign character read is saved in the local variable SIGN.

The routine continues reading in individual characters until a valid decimal character (Ø through 9) is read. the validity of characters is checked by the function routine VALID_DECIMAL_CH. The first valid decimal character received becomes the first character stored in the local array INPUT_STRING. READ_DINTEGER continues reading in characters. The reading process stops with the first invalid decimal character or when a total of five decimal charactes have been read. When an invalid character is read or after five valid characters have been read, a blank is inserted into INPUT_ STRING. The valid decimal charactes are insterted into INPUT_STRING in the order they are received.

READ_DINTEGER then enters its third phase, the translation of SIGN and INPUT_STRING into the return parameter NUMBER, of type Integer. The bulk of the work is done by general routine VALUE_LOOP which translates the characters of INPUT_STRING into the base ten value they represent. This value is checked for over- flow and then, if SIGN is "-", the value is negated. NUMBER is then returned to the calling routine.

5. Routine Relationships Diagram

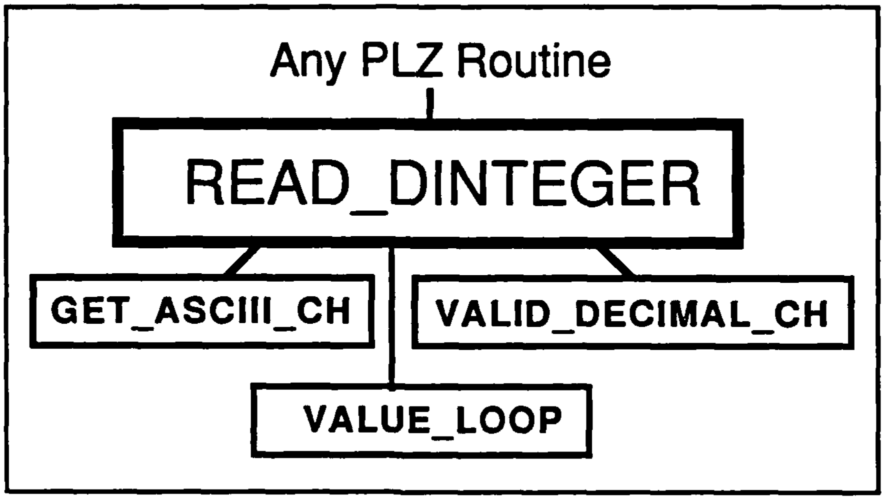


Figure 28. Relationship of READ_DINTEGER to Other Routines.

6. Invocation

a. Invocation Statement

READ_DINTEGER is invoked from a calling PLZ program through:

NUMBER := READ_DINTEGER(LOGICAL_UNIT)

where LOGICAL_UNIT is of type Byte and NUMBER is type Integer.

b. Parameter Passing Schema

LOGICAL_UNIT is READ_DINTEGER's only input parameter. It holds the number of the device the characters are input from.

c. Routines Which Call

READ_DINTEGER can be employed by any PLZ program linked with the Enhancements Module and the PLZ STREAM.IO Module.

7. Variables and Constants

a. Global

No global variables or constants are used by READ_DINTEGER.

b. Module

READ_DINTEGER uses the Enhancements Module constants BLANK, TRUE, and FALSE. No module level variables are used.

c. Routine

READ_DINTEGER has four internal variables. INPUT_STRING, type ASCII_STR (a string of 8 bytes) is used to hold the input characters. INDEX, type Byte, is a placekeeper for the array INPUT_STRING. CHARACTER, type Byte, hold each character as they are read in. Lastly, SIGN, type Byte, holds the character representing the sign of the input string. READ_DINTEGER uses no locally defined constants.

8. Other Routines Called

READ_DINTEGER employes three internal routines from the Enhancements Module, GET_ASCII_CH, VALID_DECIMAL_CH, and VALUE_LOOP.

a. GET_ASCII_CH

This routine reads single characters in from a specified logical unit and returns them to the calling routine. GET_ASCII_CH returns only valid ASCII characters. The routine is invoked by:

CHARACTERS := GET_ASCII_CH(LOGICAL_UNIT)

where both CHARACTER and LOGICAL_UNIT are of type Byte.

b. VALID_DECIMAL_CH

This function routine determines whether a character is a Ø though 9. If yes, VALID_DECIMAL_CH returns with a value of TRUE. Otherwise VALID_

DECIMAL_CH returns with a value of FALSE. The routine is invoked with:

VALID_DECIMAL_CH(CHARACTER)

where both CHARACTER and the returning VALIDE_DECIMAL_CH are type Byte.

c. VALUE_LOOP

VALUE_LOOP translates a string of ASCII characters into the value they represent and returns that value to the calling routine. Being a general purpose routine, VALUE_LOOP must be told what base the representation is in. In general, VALUE_ LOOP is invoked by:

MAGNITUDE := VALUE_LOOP(INPUT_STRING, MULTIPLIER)

where NUMBER is type Word, INPUT_STRING is type pointer-to-Byte, and MULTIPLIER is type Word. As the output of READ_DINTEGER is of type Integer, for READ_ DINTEGER, VALUE_LOOP is invoked with a type conversion. As READ_DINTEGER is converting a decimal string, MULTIPLER is passed in as 1Ø for the base.

9. Output of Routine

a. Parameter Passing Schema

READ_DINTEGER has a single return parameter, NUMBER (type Integer), which holds the value translated from the string of input characters.

b. System Configuration Changes

No system configuration changes are caused by READ_DINTEGER.

10. Routine Testing

a. Description of Test

READ_DINTEGER was tested with a version of TEST_IT module. In this version, READ_DINTEGER was called from TEST_IT to read an decimal integer in from the system keyboard. The return from READ_DINTEGER was then output to the system console. Thus the operator could input a variety of characters and observe the response of READ_DINTEGER.

b. Results of Test

READ_DINTEGER performed as expected.

11. Reference to Listing

The listing of READ_DINTEGER is on pages 304 - 305 in Appendix A.

1. Routine Name: **READ_HWORD**

2. Output routine of Enhancements Module.

3. Written in PLZ; 14 lines of executable code.

4. Synopsis of Routine

READ_HWORD is an Enhancements module routine whose function is to input a sequence of characters and translate that that sequence into the hexidecimal value it represents. The routine begins by reading in characters, one by one, until a valid hexidecimal character is received. The input is handled by the routine GET_ASCII_CH. The characters are checked by VALID_HEX_CH to determine whether the character is a Ø to9 or A to F. Once a valid hexidecimal is received, it is the first character stored in the internal array INPUT_STRINT. READ_HWORD then continues reading in chracters, one by one. Each successive valid hexidecimal character is stored in INPUT_STRING until four character are stored. If a nonvalid character is read, a blank is placed in INPUT_STRING and input is ended.

READ_HWORD next proceeds to translating the characters stored in INPUT_STRING into the hexidecimal value they represent. The work is done by routine VALUE_LOOP. The derived 16 bit value is returned to the calling routine in the output parameter NUMBER, type Word.

5. Routine Relationships Diagram

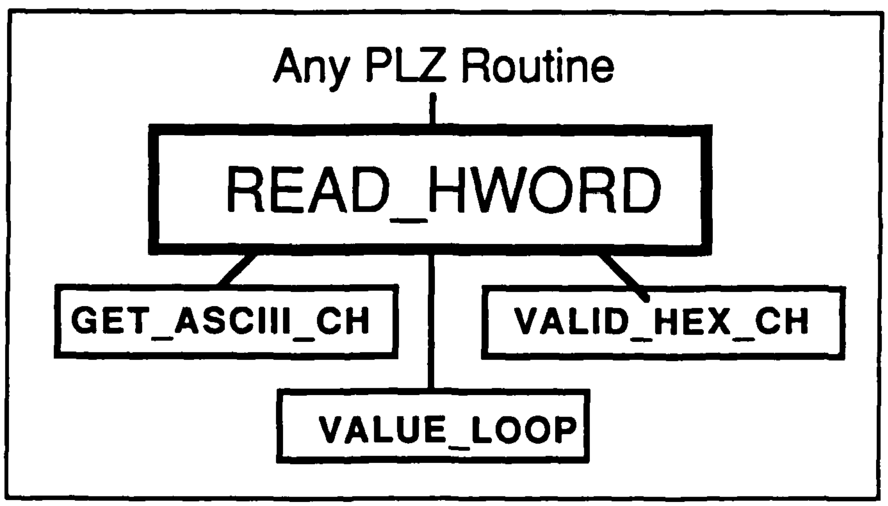


Figure 29. Relationship of READ_HWORD to Other Routines.

6. Invocation

a. Invocation Statement

From a PLZ program, READ_HWORD is invoked by:

NUMBER := READ_HWORD(LOGICAL_UNIT)

where NUMBER is type Word and LOGICAL_UNIT is type Byte.

b. Parameter Passing Schema

READ_HWORD has a single input parameter, LOGICAL_UNIT. LOGICAL_ UNIT, type Byte, holds the number of the device from which the characters are read.

c. Routines Which Call

Like the rest of the Enhancements Module routines, READ_HWORD is an supplement routine ment to ease the IO burden on PLZ programmers. READ_ HWORD can be called from any PLZ program linked with the Enhancements Module and the PLS STREAM.IO Module.

7. Variables and Constants

a. Global

No global variables or constants are used by READ_HWORD.

b. Module

READ_HWORD uses the Enhancements moudle constants TRUE and FALSE for logical true and false. No module level variables are used.

c. Routine

Three variables are local to READ_HWORD, INPUT_STRING (type ASCII_ STRing), INDEX (type Byte), and CHARACTER (type Byte). INPUT_ STRING is an eight Byte array used to hold the up to five characters (four hex characters and a blank) read in. INDEX is a place keeper for the current location being used in INPUT_STRING. CHARACTER receives the individual characters read in via GET_ASCII_CH. READ_HWORD uses no locally defined constants.

8. Other Routines Called

READ_HWORD uses three internal routines of the Enhancements Module: GET_ASCII_CH, VALID_HEX_CH, and VALUE_LOOP.

a. GET_ASCII_CH

This routine reads in a single character from a specified logical unit, checks to ensure the character is valid ASCII, and returns the character to the calling routine. GET_ASCII_CH keeps reading in data until a valid ASCII character is received. GET_ASCII_CH is invoked with:

CHARACTER := GET_ASCII_CH(LOGICAL_UNIT)

where CHARACTER and LOGICAL_UNIT are both type Byte. The input parameter, LOGICAL_UNIT, indicates the device to be used for input. CHARACTER, the return parameter, holds the valid ASCII character read in from the LOGICAL_UNIT.

b. VALID_HEX_CH

This Enhancements module internal function format routine, checks whether a character is a Ø to 9 or A to F. If yes, VALID_HEX_CH returns to the calling routine with a value of TRUE. If the character passed to VALID_HEX_CH is not a valid hexidecimal character, VALID_HEX_CH returns to the calling routine as FALSE. VALID_HEX_CH is invoked through:

VALID_HEX_CH(LOGICAL_UNIT)

where LOGICAL_UNIT, type Byte, identifies the device from which data is to be read.

c. VALUE_LOOP

VALUE_LOOP is a general purpose translation routine. It takes a string of characters, in any base from 2 to 16, and translates the string into the value they represent. VALUE_LOOP is invoked by:

MAGNITUDE := VALUE_LOOP(INPUT_STRING, MULTIPLER) where MAGNITUDE (type Word) is the value represented by the characters, INPUT_ STRING (type pointer-to-string) is the string of characters, and MULTIPLIER (type Word) is the base of the character representation.

9. Output of Routine

READ_HWORD has a single output parameter, NUMBER, of type Word. NUMBER holds value translated from the input characters. The defined range of NUMBER is ØØØØ to FFFF hexidecimal. READ_HWORD causes no configuration changes.

10. Routine Testing

a. Description of Test

READ_HWORD was tested through a routine in version of TEST_IT Module which uses READ_HWORD to read in characters from the keyboard and translate them into a value. This value is then displayed to the system console. This way, the function of READ_HWORD can be immediately observed.

b. Results of Test

READ_HWORD worked properly.

11. Reference to Listing

READ_HWORD's program listing is on page 306 in Appendix A.

1. Routine Name: **READ_DWORD**

2. Output routine of Enhancements Module.

3. Written in PLZ; 14 lines of executable code.

4. Synopsis of Routine

READ_DWORD is an Enhancements Module routine whose function is to input a sequence of characters and translate that that sequence into the decimal value it represents. The routine begins by reading in characters, one by one, until a valid decimal character is received. The input is handled by the routine GET_ASCII_CH. The characters are checked by VALID_DECIMAL_CH to determine whether the character is a Ø to9. Once a valid decimal is received, it is the first character stored in the internal array INPUT_STRING. READ_DWORD then continues reading in chracters, one by one. Each successive valid decimal character is stored in INPUT_STRING until six character are stored. If a nonvalid character is read, a blank is placed in INPUT_STRING and input is ended.

READ_DWORD next proceeds to translating the characters stored in INPUT_STRING into the decimal value they represent. The work is done by routine VALUE_LOOP. The derived 16 bit value is returned to the calling routine in the output parameter NUMBER, type Word.

5. Routine Relationships Diagram

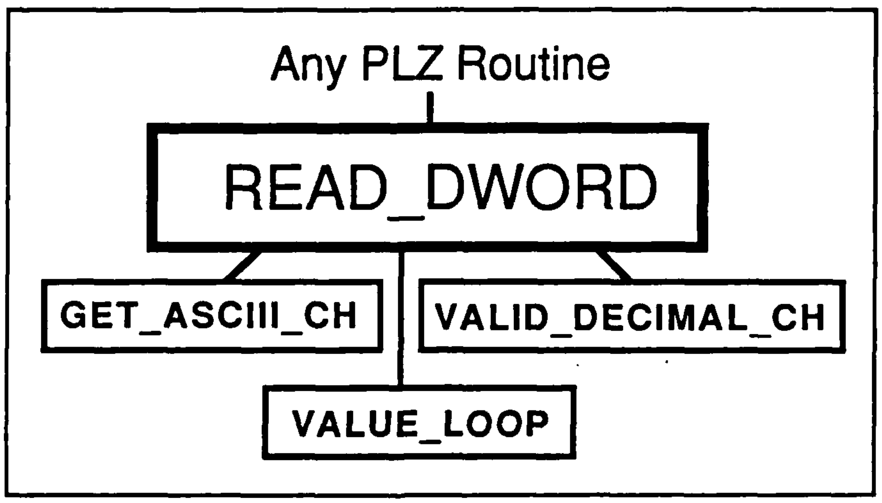


Figure 30. Relationship of READ_DWORD with Other Routines

6. Invocation

a. Invocation Statement

From a PLZ program, READ_DWORD is invoked by:

NUMBER := READ_DWORD(LOGICAL_UNIT)

where NUMBER is type Word and LOGICAL_UNIT is type Byte.

b. Parameter Passing Schema

READ_DWORD has a single input parameter, LOGICAL_UNIT. LOGICAL_ UNIT, type Byte, holds the number of the device from which the characters are read.

c. Routines Which Call

Like the rest of the Enhancements module routines, READ_DWORD is a supplement routine ment to ease the IO burden during PLZ programming. READ_ DWORD can be called from any PLZ program linked with the Enhancements module and the PLS STREAM.IO module.

7. Variables and Constants

a. Global

No global variables or constants are used by READ_DWORD.

b. Module

READ_DWORD uses the Enhancements module constants TRUE, FALSE, and BLANK. No module level variables are used.

c. Routine

Three variables are local to READ_DWORD, INPUT_STRING (type ASCII_ STRing), INDEX (type Byte), and CHARACTER (type Byte). INPUT_ STRING is an eight Byte array used to hold the up to seven characters (six decimal characters and a blank) read in. INDEX is a place keeper for the current location being used in INPUT_STRING. CHARACTER receives the individual characters read in via GET_ASCII_CH. READ_ DWORD uses no locally defined constants.

8. Other Routines Called

READ_DWORD uses three internal routines of the Enhancements Module: GET_ASCII_CH, VALID_DECIMAL_CH, and VALUE_LOOP.

a. GET_ASCII_CH

This routine reads in a single character from a specified logical unit, checks to ensure the character is valid ASCII, and returns the character to the calling routine. GET_ASCII_CH keeps reading in data until a valid ASCII character is received. GET_ ASCII_CH is invoked with:

CHARACTER := GET_ASCII_CH(LOGICAL_UNIT)

where CHARACTER and LOGICAL_UNIT are both type Byte. The input parameter, LOGICAL_UNIT, indicates the device to be used for input. CHARACTER, the return parameter, holds the valid ASCII character read in from the LOGICAL_ UNIT.

b. VALID_DECIMAL_CH

This Enhancements Module internal function format routine, checks whether a character is a Ø to 9. If yes, VALID_DECIMAL_CH returns to the calling routine with a value of TRUE. If the character passed to VALID_DECIMAL_CH is not a valid decimal character, VALID_DECIMAL_CH returns to the calling routine as FALSE. VALID_DECIMAL_CH is invoked through:

VALID_DECIMAL_CH(LOGICAL_UNIT)

where LOGICAL_UNIT, type Byte, identifies the device from which data is to be read.

c. VALUE_LOOP

This Enhancements Module routine is a general purpose translation routine. It takes a string of characters, in any base from 2 to 16, and translates the string into the value they represent. VALUE_LOOP is invoked by:

MAGNITUDE := VALUE_LOOP(INPUT_STRING, MULTIPLER)

where MAGNITUDE (type Word) is the value represented by the characters, INPUT_ STRING (type pointer-to-string) is the string of characters, and MULTIPLIER (type Word) is the base of the character representation. VAUE_LOOP performs a crude overflow checking and returns the maximum 16 bit value (65,535 decimal) if overflow is detected.

9. Output of Routine

READ_DWORD has a single output parameter, NUMBER, of type Word. NUMBER holds value translated from the input characters. The defined range of NUMBER is Ø to 65,535 decimal. READ_DWORD causes no configuration changes.

10. Routine Testing

a. Description of Test

READ_DWORD was tested through a routine which used READ_DWORD to read decimal characters in from the keyboard and translate them into a value. This value was then displayed to the system console. Thus, the function of READ_DWORD was immediately observed.

b. Results of Test

READ_DWORD worked properly.

11. Reference to Listing

The listing of routine READ_DWORD is on page 307 in Appendix A.

This page is intentionally blank

III. Utility Module

Introduction to Utility Module

Utility Module is a collection of nine Z-8Ø assembly language routines designed to give PLZ language programs direct access to input/output ports, specific memory locations, the CPU interrupt enable/disable, the system date, and the RIO Operating System memory manager. These assembly language routines are called as subroutines from PLZ programs. The routines of the Utility Module and their functions are:

IOOUT:	Outputs desired value to desired IO port.
IOIN:	Reads input from desired IO port.
MEMSET:	Writes a desired value to a specific memory cell.
MEMREAD:	Reads the value stored in a specific memory cell.
DISABLEINT:	Disables the CPU maskable interupts
ENABLEINT:	Enables the CPU maskable interupts
DATE:	Reads the six characters of the system date.
ALLOCATE:	Calls the memory manager for allocation of a specific sized block of memory.
DEALLOCATE:	Calls the memory manager for the deallocation of a specific block of memory.

Figure 31 below shows how these nine routines relate to the their calling PLZ routines and to elements of the development system.

Seven of the nine Utility Module routines share several common features mandated by the PLZ subroutine call and parameter passing procedures (Ref 6:Sec 7). These features are:

1. Saving the current IX register value,
2. Placing the stack pointer value in the IX register and using offsets for access to input and output parameters,

3. Code to accomplish the routine's specific task,

4. Restoring of calling routine's IX register value,
5. Deallocating of input parameter space on the stack, and
6. Returning to the calling PLZ routine.

The program listings for the seven routines are organized like the above feature listing with blank lines setting off the PLZ overhead from the routine's function code. Two routines of the Utility Module, ENABLEINT and DISABLEINT do not share the common fea- tures listed above. This is due to their lack of input and output parameters and their sim- plicty; they just do not require the overhead of the other seven routines. The reasons for and the form of this overhead of the other seven routines is detailed below.

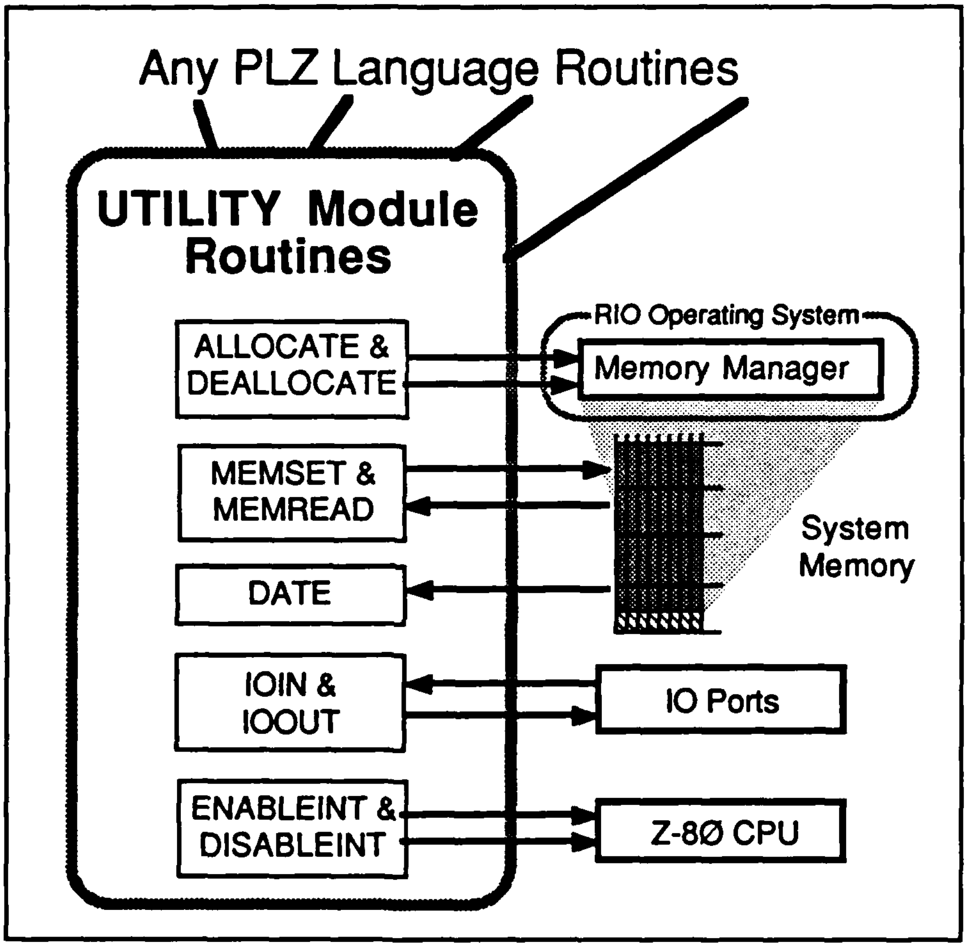


Figure 31. Relationship Between the Routines of the Utility Module to Calling Routines and System Elements.

The first action of the seven routines is to save the current value of the IX register by pushing it onto the stack This is vital. The IX, upon entry to the Utility Module routine, points to the calling routine's parameters and must be restored if the overall pro- gram is to properly execute upon return from the Utility Module routine. ENABLEINT and DISABLEINT do use the IX register and thus do not have to save its contents.

Next, the current stack pointer value is placed in the IX register. Offsets from IX will be used throughout the routines to access and save input and output parameters respectively. PLZ uses a table called an Activation Record (AREC) to pass parameters between calling routines and subroutines. The AREC is placed on the system stack by the calling routine. The AREC contains (from high address to low address):

1. Output or Return Parameters,
2. Input Parameters,
3. Local Parameters (of the called routine), and
4. A Mark-Stack Record (MREC) consisting of the return address of the calling routine and the IX register value of the calling routine.

For the Utility Module routines, there are no local parameters. Upon entry to the called routine, the stack pointer will point to the low memory boundry of the AREC. Thus by loading it into the IX register, offsets can be easily used.

The amount of offset from the IX depends upon the number and size of values present in the AREC (Ref 6:7-2). For example, variables of PLZ type Byte require only one location (one byte) in the AREC while variables of type Word require two locations. Return parameters however, are always passed in sixteen bit forms, reguardless of type. Strings are handled by passing pointers (sixteen bits) to the beginning of the string. Figure 32 below gives an example of the AREC for ALLOCATE, the Utility Module routine having the most complex set of parameters. Note that the return parameter RETURN_ CODE is passed in a sixteen bit space dispite it being of type byte and requiring only eight bits.

The third section of the seven Utility Module routines is the unique code of each routine uses to accomplish its function. While different in purpose, the code of the seven Utility Module routines share the use of offsets from the IX register to access input parameters and to load output parameters.

The fourth common feature of the seven routines is the restoration of the calling routine's IX register through a POP IX instruction. This instruction flags the end of the routine function code and the beginning of the final three PLZ overhead management steps.

The next to last step is the deallocation of input parameter (in PLZ parlance these are out parameters) storage space on the stack. As with saving the IX value, this action is vital. Upon return to the calling routine, PLZ expects to find the return, local, and all other parameters needed by the calling routine as off sets from the stack pointer. It the deallocation of input parameters isn't accomplished, all the stack pointer offsets will be invalid. ENABLEINT and DISABLEINT do not go through this step as they do not have input parameters. However, all the routines do pop into the HL register the return address of the calling routine.

The sixth and final step, common to all nine Utility Module routines, is the return to the calling PLZ routine. This is accomplished simply by a JP (HL) for jump to the address in the HL register instruction, the return address having already been popped into the HL register. With that action, the Utility Module routine ends.

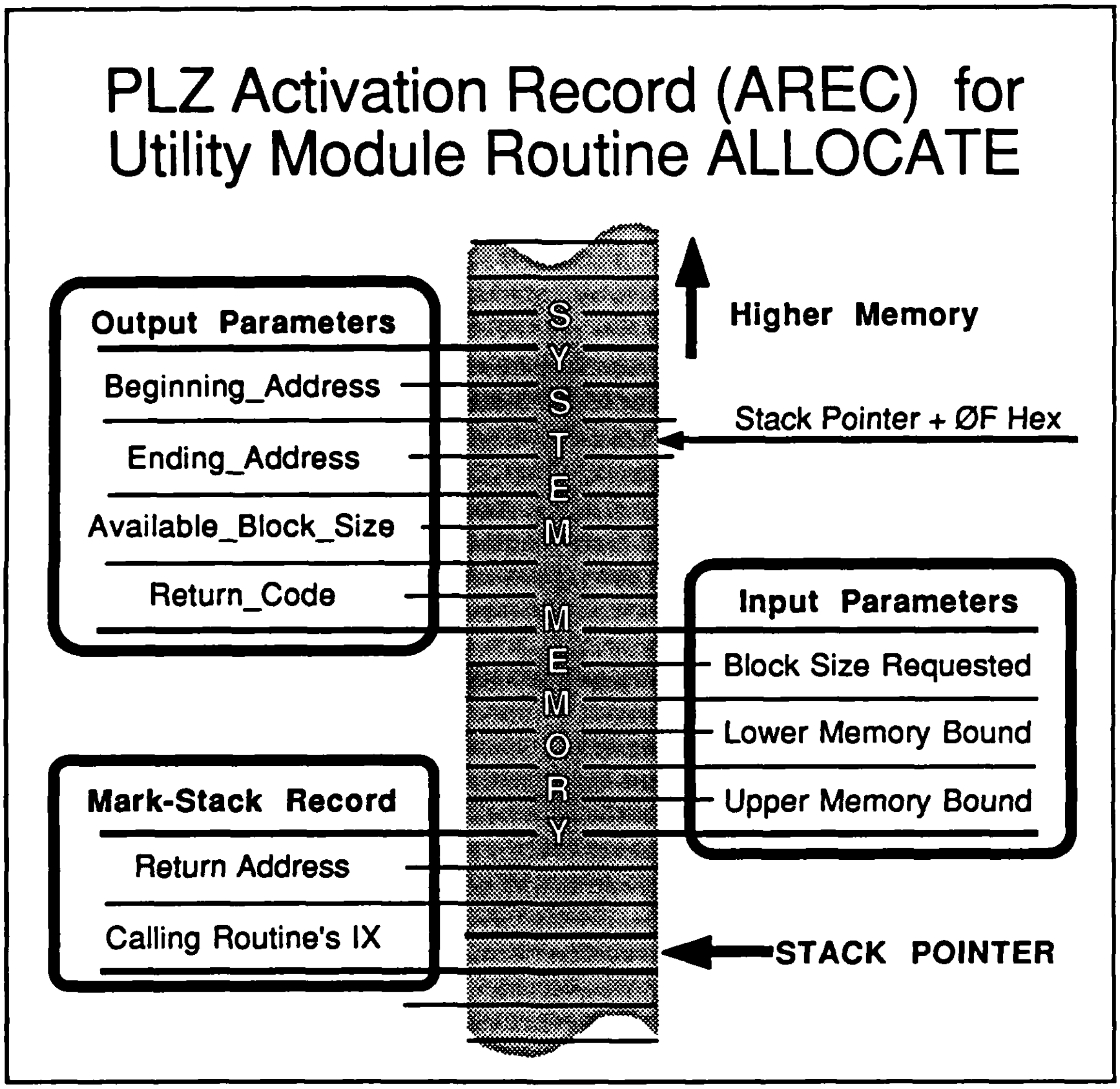


Figure 32. Example of PLZ Activation Record — ALLOCATE.

Thus, the nine assembly language routines of the Utility Module give PLZ language routines direct access to input/output port, system memory, the system date, the Z-8Ø interrupt enable/disable, and the RIO operating system memory management routine. The following pages detail the nine routines. For each routine the following information will be presented.

1. The name of the routine.
2. The name of the routine's module.
3. The language the routine is written in and the number of lines of code in the routine.
4. A synopsis of the routine.
5. A Routine Relationship Diagram showing the relationship of the routine to the PLZ routines that call it and the elements of the system that it calls.
6. The invocation statement for the routine, its input parameter passing schema, and the routines called by the routine.
7. A description of the global, module, and local level constants used by the routine.
8. Descriptions, including parameter passing, of all routines called by the routine.
9. A discussion of the output of the routine, both output parameters and effect on system configuration.
10. The testing of the routine.
11. A reference to the pages of the routine code listing.

The code listing of the routines of Utility Module are in Appendix B.

1. Name: **IOOUT**

2. Part of Utility Module

3. Written in Z-8Ø Assembly Language; 22 bytes.

4. Synopsis of Routine

IOOUT is an assembly language routine which gives PLZ language routines direct access to the input/output ports of the system. Through IOOUT a PLZ program can write directly to the output registers. IOOUT has three sections of code, AREC save, write to IO port, and return to calling routine.

5. Routine Relationship Diagram

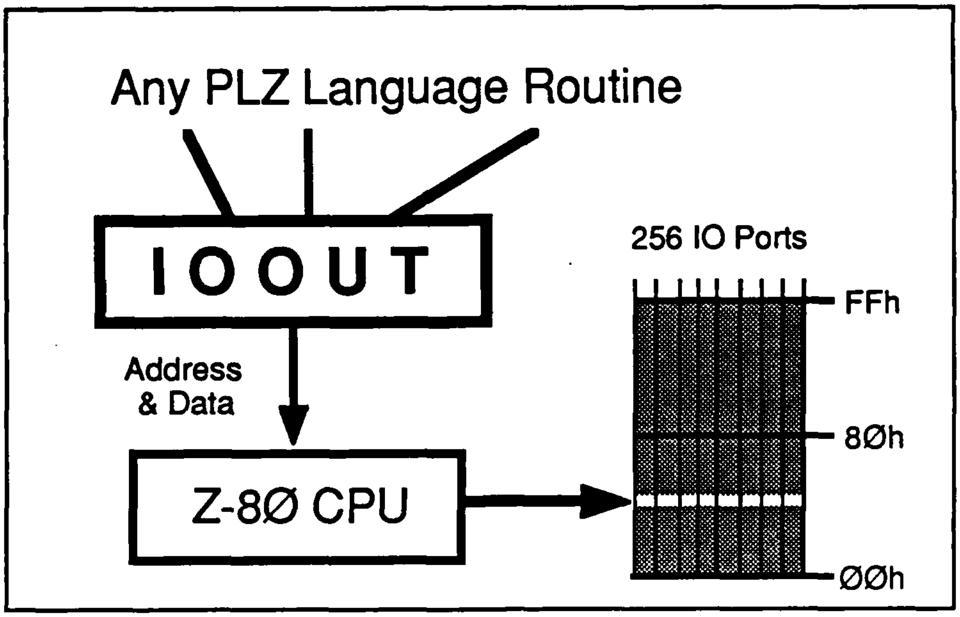


Figure 33. Relationship of IOOUT to Calling PLZ Routines and the Central Processing Unit

6. Invocation

a. Invocation Statement

IOOUT is invoked in a PLZ routine via:

IOOUT(IO_PORT, VALUE)

where both IO_PORT and VALUE are of type BYTE.

b. Input Parameter Passing Schema

IOOUT has two input parameters, IO_PORT and VALUE, both of type Byte. IO_PORT is the number of the input/output port to which the data will be output. The defined rage of IO_PORT is 0 to 255. VALUE is the quantity to be output to the designated IO_PORT.

c. Routines Which Call

Though IOOUT call be called by any PLZ routine, it is not used by any of the final software in this thesis effort.

7. Variables and Constants

a. Global

IOOUT uses no global constants or variables outside the defined uses of the IX and HL registers for subroutine entry / exit.

b. Internal to the Module

IOOUT uses the module constant ZERO, value ØØØØ hex. IOOUT uses no module level variables.

c. Internal to the Routine

None

8. Other Routines Called

IOOUT calls no other routines.

9. Output of Routine

The output of IOOUT is the writing of the desired VALUE to the desired IO_PORT. There are no other effects. IOOUT has no output parameters.

10. Routine Testing

a. Description of Test

IO_OUT was tested with a simple PLZ routine which writes predetermined values to predetermined IO Ports. The ports were monitored with a logic analyzer.

b. Results of Test

The desired values were written to the proper ports. Conclusion: IOOUT works.

11. Reference to Listing

The listing of IOOUT is on page 317 in Appendix B.

1. Routine Name: **IOIN**

2. Part of Utility Module

3. Written in Z-8Ø Assembly Language; 25 bytes.

4. Synopsis of Routine

IOIN is an assembly language routine which gives a PLZ language routine direct access to the input/output ports of the system. Through IOIN a PLZ program can directly read from the IO ports. IOIN has three subdivisions, AREC save, IO port read, and return to calling routine.

5. Routine Relationship Diagram

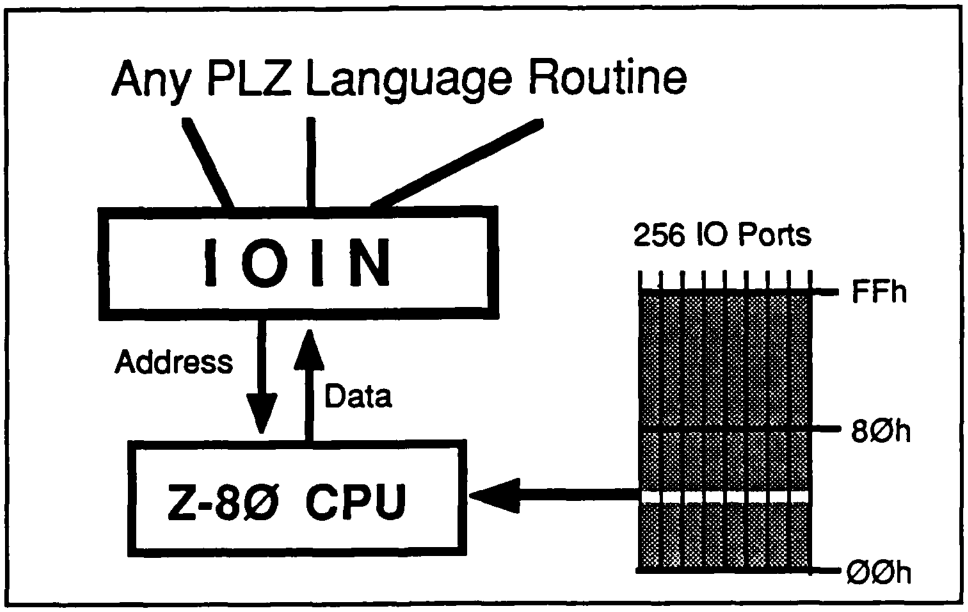


Figure 34. Relationship of IOIN to Calling PLZ Routines and the Central Processing Unit.

6. Invocation

a. Invocation Statement

IOIN is invoked in a PLZ routine via:

VALUE := IOIN(IO_PORT)

where both IO_PORT and VALUE are of type BYTE.

b. Input Parameter Passing Schema.

IOIN has one input parameter, IO_PORT, the number of the input / output port data is to be read from.

c. Routines Which Call IOIN

IOIN can be called by any PLZ routine. In this thesis effort IOIN was not used in the final software.

7. Variables and Constants

a. Global

IOIN uses no global constants or variables outside the defined uses of the IX and HL registers for subroutine entry/exit.

b. Internal to the Module

The module constant ZERO, value ØØØØ hex, is used by IOIN; there are no module level variables.

c. Internal to the Routine

None

8. Other Routines Called

IOIN calls no other routines.

9. Output of Routine

a. Output Parameter Passing Schema.

IOIN has one output parameter, VALUE (type Byte), which holds the data read in from the IO port indicated by the input parameter IO_PORT.

b. System Configuration Changes

Beyond the impact of the read upon the IO port's status, IOIN causes no system changes.

10. Routine Testing

a. Description of Test

IOIN was tested with a simple PLZ routine which read (via IOIN) from a serial IO port which was connected to a terminal. Characters were typed in at the terminal. The characters (VALUEs) read were then displayed to the system console.

b. Results of Test

The characters typed in at the terminal appeared on the system console. *Conclusion:* IOIN works.

11. Reference to Listing

The program listing for IOIN is located on page 318 in Appendix B.

1. Name: **MEMSET**

2. Part of Utility Module

3. Written in Z-8Ø Assembly Language; 24 bytes.

4. Synopsis of Routine

MEMSET is an assembly language routine which permits PLZ language routines to write to or set specific random access memory (RAM) locations to specific values. MEMSET's code has three major subdivisions: AREC save, write to a memory location, and return to calling routine.

5. Routine Relationship Diagram

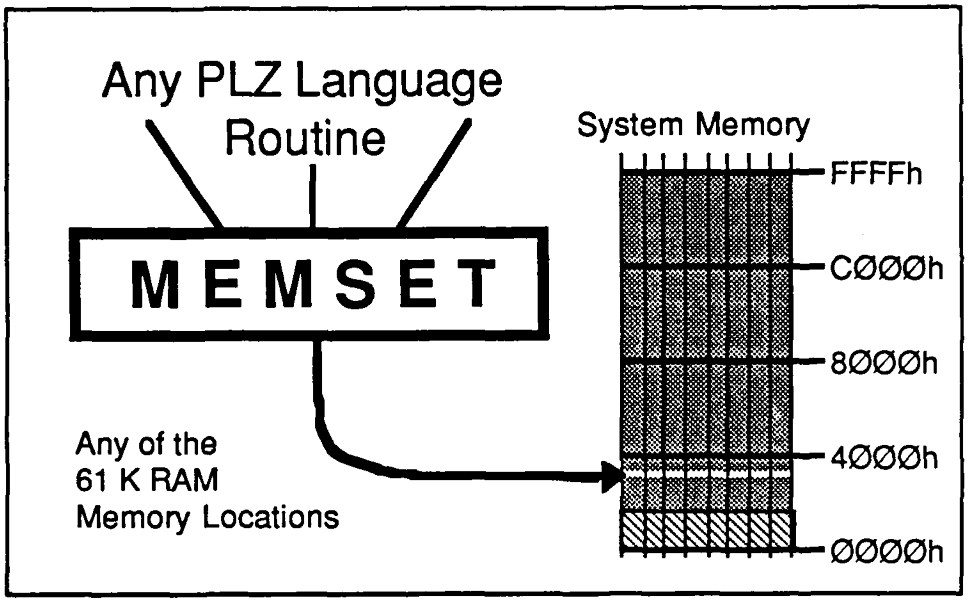


Figure 35. Relationship of MEMSET to Calling PLZ Routines.

6. Invocation

a. Invocation Statement

MEMSET is invoked in a PLZ program via:

MEMSET(LOCATION, VALUE)

where LOCATION is type Word and VALUE is type byte.

b. Input Parameter Passing Schema

MEMSET has two input parameters, LOCATION (type Word), the address of specific memory location, and VALUE (type Byte), the quantity to be stored in the location. These parameters are passed via standard PLZ methods.

c. Routines Which Call MEMSET

MEMSET can be used by any PLZ routine needing direct access to memory locations. MEMSET was not used by the final routines of this thesis effort.

7. Variables and Constants

a. Global

No internal module variables or constants besides the registers used by PLZ subroutine calls.

b. Internal to the Module

MEMSET uses the constant ZERO of value ØØØØ Hex; no module level variables are used.

c. Internal to the Routine

MEMSET uses two of the CPU registers to hold variables. The HL register holds the address of memory location to be read and the A register holds the value read from memory location. No routine level constants are used.

8. Other Routines Called

MEMSET calls no other routines.

9. Output of Routine

a. Output Parameter Passing Schema

MEMSET has no output parameters.

b. System Configuration Changes

MEMSET changes the quantity stored in the desired memory location to the specified value.

10. Routine Testing

a. Description of Test

MEMSET was tested by having a simple PLZ routine setting specific memory locations to know values via MEMSET. Then the debugger was used to display the same memory locations.

b. Results of Test

MEMSET set the proper memory locations to the proper values. Conclusion: MEMSET works.

11. Reference to Listing

The program listing of MEMSET is on page 319 in Appendix B.

1. Name: **MEMREAD**

2. Part of Utility Module

3. Written in Z-8ØAssembly Lanugage; 27 bytes.

4. Synopsis of Routine

MEMREAD is an assembly language routine which permits PLZ language routines to read specific memory locations, RAM OR ROM. MEMREAD has three major subdivisions: AREC save, read of memory location, return to calling routine.

5. Routine Relationship Diagram

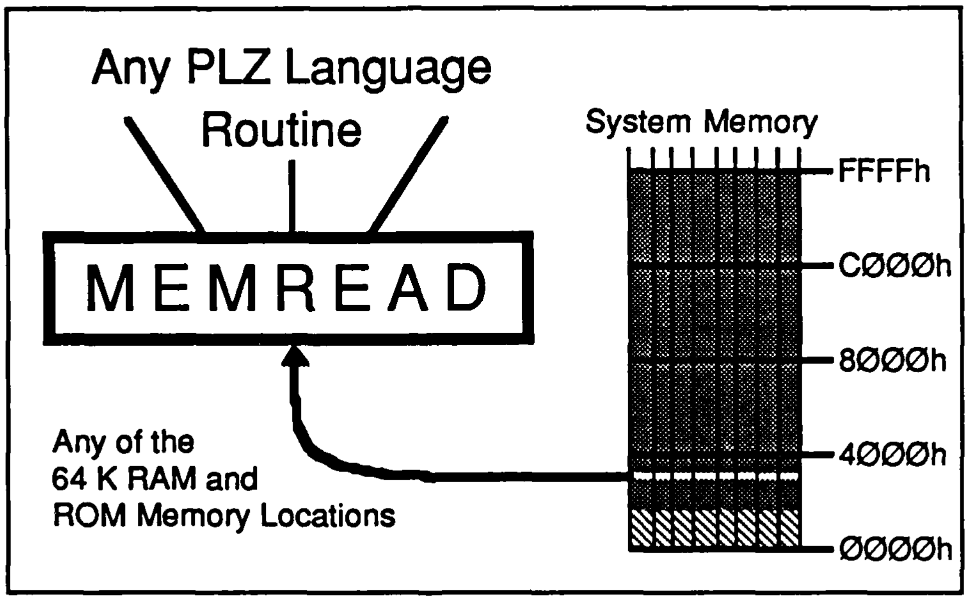


Figure 36. Relationship of MEMREAD to Calling PLZ Routines.

6. Invocation

a. Invocation Statement

MEMREAD is invoked in the calling PLZ routine via:

VALUE := MEMREAD(LOCATION)

where VALUE is of type Byte and LOCATION is of type Word.

b. Input Parameter Passing Schema

MEMREAD has one input parameter, LOCATION (type Word), the address of specific memory location to be read. LOCATION has a defined range of Ø to 65535 decimal.

c. Routines Which Call MEMREAD

MEMREAD was not used by any of the final data collection routines of this thesis effort. However, it can be used by any PLZ language routine needing direct access to memory.

7. Variables and Constants

a. Global

MEMREAD uses no global constants or variables.

b. Internal to the Module

Besides the registers used by PLZ subroutine calls, MEMREAD uses no module level variables. The module constant ZERO, value ØØØØ hex, is used by MEMREAD.

c. Internal to the Routine

MEMREAD employs two CPU registers to hold variables. The HL register holds the address of memory location to be read and the A register holds the value read from memory location. There are no routine level constants.

8. Other Routines Called

MEMREAD calls no other routines.

9. Output of Routine

a. Output Parameter Passing Schema

MEMREAD has one output parameter, VALUE, which is the quantity stored in the memory location specified by the input parameter LOCATION.

b. System Configuration Changes

MEMREAD causes no system changes.

10. Routine Testing

a. Description of Test

MEMREAD was tested by setting memory locations to know values with the debugger. Then a simple PLZ routine, which reads the same memory locations (via MEMREAD) and displays them on the console, was run.

b. Results of Test

The values stored in the memory locations were properly displayed. Conclusion: MEMREAD works.

11. Reference to Listing

MEMREAD's program listing is on page 320 in Appendix B.

1. Name: **DISABLEINT**

2. Part of Utility Module

3. Written in Z-8Ø Assembly Language. Three bytes.

4. Synopsis of Routine

DISABLEINT is a very simple assembly language routine which enables a PLZ routine to disable the Z-8Ø interrupts. This routine is a companion to ENABLEINT.

5. Routine Relationship Diagram

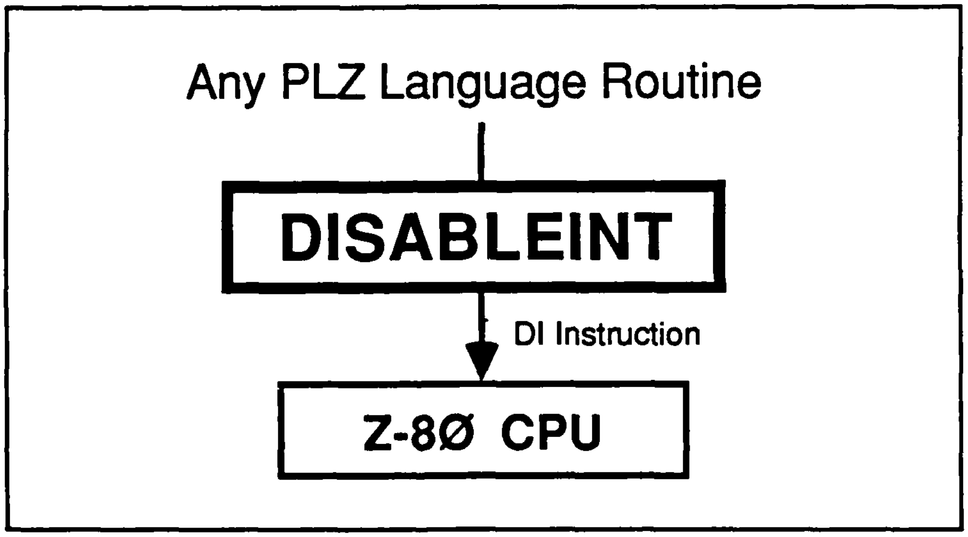


Figure 37. Relationship of DISABLEINT to Calling PLZ Routines and the Interrupt Setting of the Central Processing Unit.

6. Invocation

a. Invocation Statement

DISABLEINT is called from a PLZ program via:

DISABLEINT

b. Parameter Passing Schema

DISABLEINT has no parameters.

c. Routines Which Call DISABLEINT

DISABLEINT was used by the AIO.PLZ.S Module routines which served as precursors for the Sampler Module assembly language routines.

7. Variables and Constants

The only "variable" used by DISABLEINT is the HL register which stores the address of the calling routine.

8. Other Routines Called

DISABLEINT calls no other routines.

9. Output of Routine

The result of DISABLEINT is the disabling of the Z-80 interrupts.

10. Routine Testing

a. Description of Test

DISABLEINT is called by another routine which causes interrupts. With that routine running, a logic analyzer was used to monitor the CPU lines.

b. Results of Test

Before the invocation of DISABLEINT the CPU responded to the interrupt signals. After the invocation of DISABLEINT the CPU ignored the interrupt signals. Conclusion: DISABLEINT works.

11. Reference to Listing

The program listing for DISABLEINT is on page 321 in Appendix B.

1. Name: **ENABLEINT**

2. Part of Utility Module

3. Written in Z-8Ø Assembly Language, three bytes.

4. Synopsis of Routine

ENABLEINT is a very simple assembly language routine which enables a PLZ routine to enable the Z-8Ø interrupts. ENABLEINT is a companion to routine DISABLEINT.

5. Routine Relationship Diagram

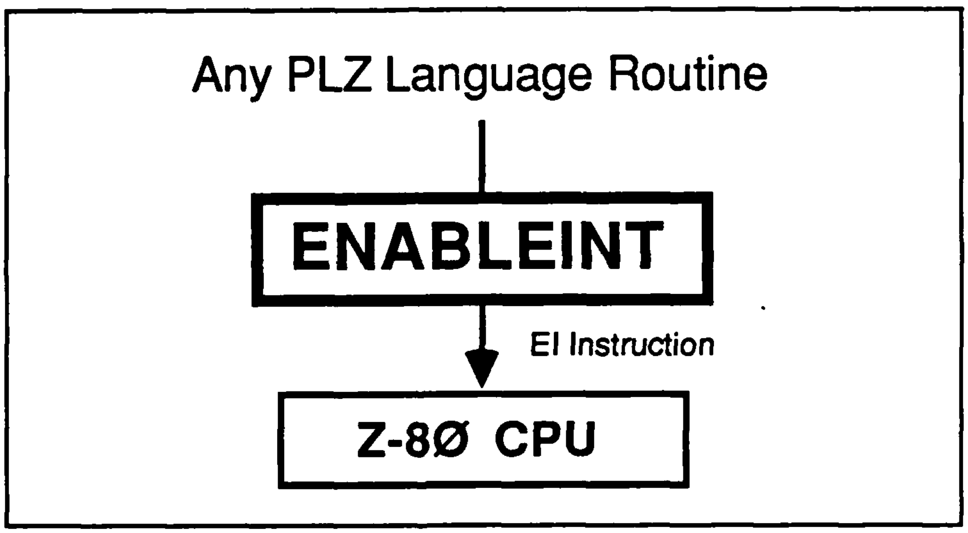


Figure 38. Relationship of ENABLEINT to Calling PLZ Routines and the Interrupt Setting of the Central Processing Unit.

6. Invocation

a. Invocation Statement

ENABLEINT is invoked from the calling PLZ routine via:

ENABLEINT.

b. Parameter Passing Schema

ENABLEINT has no parameters.

c. Routines Which Call ENABLEINT

This routine was not used by any of the final version routines of the data collection system. However, ENABLEINT was used by the AIO.PLZ.S Module during initial software design.

7. Variables and Constants

The only "variable" used by ENABLEINT is the HL register which stores the address of the calling routine.

8. Other Routines Called

ENABLEINT calls no other routines.

9. Output of Routine

The result of ENABLEINT is the enabling of the Z-8Ø interrupts.

10. Routine Testing

a. Description of Test

ENABLEINT is called by another routine which uses interrupts. With that routine running, a logic analyzer was used to monitor the CPU lines.

b. Results of Test

Prior to the invocation of ENABLEINT the CPU ignored the interrupt signals; afer invocation, the interrupts were acknowledged. Conclusion: ENABLEINT works.

11. Reference to Listing

The listing of ENABLEINT is on page 322 in Appendix B.

1. Routine Name: **DATE**

2. Part of Utility Module

3. Written in Z-8Ø Assembly Language; 33 bytes.

4. Synopsis of Routine

Procedure DATE is an assembly language routine which permits a PLZ language routine to call the operating system an obtain the current system date. DATE has four major subdivisions.

First, DATE saves the IX register for later restoration.

Second, DATE prepairs pointers to both the stack and the memory locations where the date is stored.

Third, DATE copies the six characters from the date storage locations to the stack.

Fourth, DATE restores the IX register, gets the return address, and returns control to the calling PLZ routine.

5. Routine Relationship Diagram

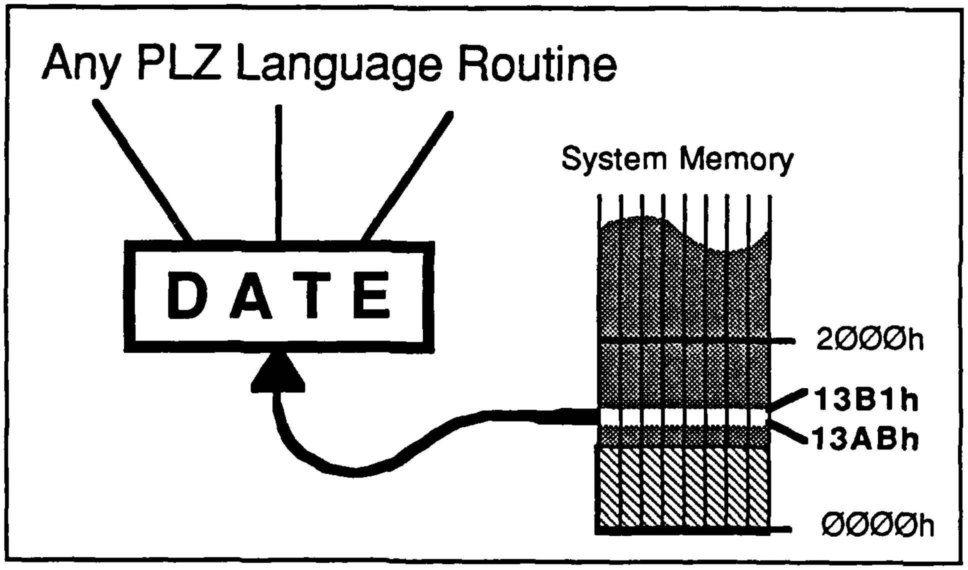


Figure 39. Relationship of DATE to Calling PLZ Routines and *Memory Locations of Date Characters.*

6. Invocation

a. Invocation Statement

DATE is invoked in the calling PLZ routine by:

YEAR1, YEARØ, MONTH1, MONTHØ, DAY1, DAYØ := DATE

where these return parameters are of single character type.

b. Input Parameter Passing Schema

DATE has no input parameters; it uses the six system date characters stored in memory locations 13AB –13CØ.

c. Routines Which Call DATE

Any PLZ program which has been linked with the Utility Module can call DATE. For this thesis effort, DATE is called by GET_DATE of the Collect_Data Module.

7. Variables and Constants

a. Global Constants

ZERO:	ØØØØ Hex, just a constant for zero
DATE_ADDRESS:	13AB Hex, the first of six system date memory locations

b. Variables Internal to the Module

Named module variables per say are not used, however, some registers of the Z-8Ø are used by the subroutine call schema. The return address is on the top of the stack at the onset of the called subroutine. The IX register is used by PLZ to point to the Activation Record (AREC), a table of pointers created for subroutine calls. Thus, it is important to save and restore the IX register.

c. Variables Internal to the Routine

Though no named variables are used, several of the Z-8Ø CPU registers are used to hold variables. The C register is used to count down the 6 character-transfers. The HL register points to the system date storage location for each character. The DE register points to the output storage location for each character, the destination location.

8. Routines Called by DATE

DATE calls no other routines.

9. Output of Routine

a. Output Parameter Passing Schema

DATE outputs six parameters, the six ASCII characters of the system date. These six parameters, YEAR1, YEARØ, MONTH1, MONTHØ, DAY1, and DAYØ, are all of type Byte.

b. System Configuration Changes

DATE does not modify any system configurations.

10. Routine Testing

a. Description of Test

DATE was tested by loading the system date (via RIO routine DATE with known values and then running a simple PLZ routine which called DATE and output the returned values to the screen.

b. Results of Test

It worked properly.

11. Reference to Listing

The listing of DATE can be found on page 323 in Appendix B.

1. Name: **ALLOCATE**

2. Part of Utility Module

3. Written in Z-8Ø Assembly Language; 82 bytes.

4. Synopsis of Routine

ALLOCATE is an assembly language routine which permits PLZ language routines access to the system memory manager. The specific purpose of ALLOCATE is memory allocation; DEALLOCATE is a companion routine. ALLOCATE has seven subdivisions.

a. AREC save
b. Load of input parameters into Registers for OS call.
c. Call to memory manager to allocate memory.
d. Load of two OS response parameters into subroutine return locations.
e. Error Code evaluation.
f. Load of remaining OS response parameters into subroutine return locations.
g. Return to calling routine.

5. Routine Relationship Diagram

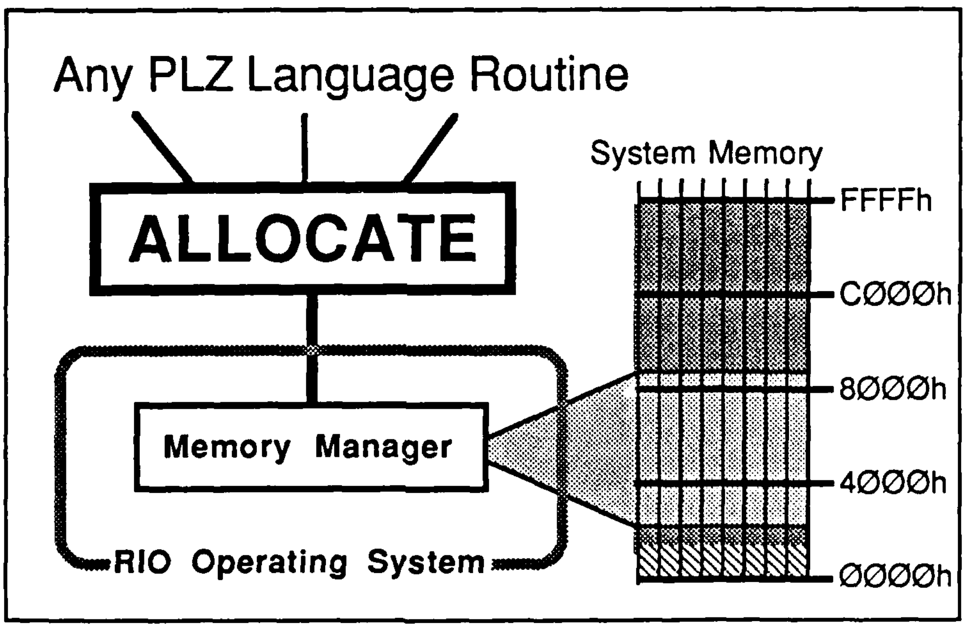


Figure 40. Relationship of ALLOCATE to Calling PLZ Routines and the RIO Operating System.

6. Invocation

a. Invocation Statement

ALLOCATE is invoked in a PLZ program as follows.

```
RETURN_CODE, AVAILABLE_BLOCK SIZE,
BEGINNING_ADDRESS, ENDING_ADDRESS :=
                              ALLOCATE( BLOCK_SIZE_REQUESTED,
                                        LOWER_MEMORY_BOUND,
                                        UPPER_MEMORY_BOUND   )
```

where RETURN_CODE is type Byte, and the remaining parameters are type Word.

b. Input Parameter Passing Schema

ALLOCATE uses three input parameters and follows the standard subroutine parameter passing methods. The input parameters are:

BLOCK_SIZE_REQUESTED: This is the size of memory block, in bytes, for which memory allocation is being requested. As this is is of type Word, its defined range is Ø to 65,536 (64K). Type Word.

LOWER_MEMORY_BOUND: The memory location that allocation must be above. Defined range Ø to 64K. This parameter is used to fence out areas of memory for other use. Type Word.

UPPER_MEMORY_BOUND: The memory location that the allocation must be below. Defined range Ø to 64K. This parameter is used to fence out areas of memory. Type Word.

c. Routines Which Call ALLOCATE

The current versions of the data collection software do not use ALLOCATE. However, ALLOCATE would be used by an improved SIZE_DAT_BUFFER (Collect_ Data Module) to provide direct access to the RIO Operating System Memory Manager.

7. Variables and Constants

a. Global

There are no true global variables or constants used by ALLOCATE.

b. Constants Internal to the Module

ZERO:	ØØØØ Hex
ALCT_MEMORY:	ØØ Hex, the code for allocate memory passed to the memory manager.
MEMORY_MANAGER:	14Ø9 Hex, the address of the memory manager entry point.
OPERATION_COMPLETE:	8Ø Hex, the return code for successful memory allocation.

c. Internal to the Routine

Besides the use of the CPU registers to hold parameters (see below), ALLOCATE has no internal constants or variables.

8. Other Routines Called

ALLOCATE calls the RIO Operating System Memory Manager. The CPU registers are used to pass parameters between ALLOCATE and the Memory Manager. For the call to the Memory Manager:

BC holds the BLOCK_SIZE_REQUESTED in bytes;
HL holds the LOWER_MEMORY_BOUND address;
DE holds the UPPER_MEMORY_BOUND address;
A holds the request code for memory allocation, ØØ hex.

The Memory Manager returns:

BC holds the AVAILABLE_BLOCK_SIZE (which may be that requested);
HL holds the BEGINNING_ADDRESS of the allocated or available block;
DE holds the ENDING_ADDRESS of the allocated or available block;
A holds the RETURN_CODE.

The values placed in the registers and returned by the memory manager are functionally the same as the input and output parameters of ALLOCATE.

9. Output of Routine

a. Output Parameter Passing Schema

The four parameters returned by ALLOCATE to the calling PLZ routine are:

RETURN_CODE: Type Byte. The return code is the operating system's message on its success in allocating the desired block of memory. If a block of memory was successfully allocated the RETURN_ CODE will be zero. On the other hand, if a contiguous block of the desired size could not be found, RETURN_CODE will have the value 4A hex which means insufficient memory.

AVAILABLE_BLOCK_SIZE: Type Word. The value returned in this parameter depends upon whether the BLOCK_SIZE_REQUESTED was available. If it was, then AVAILABLE_BLOCK_SIZE is the number of bytes requested. If however the BLOCK_SIZE_REQUESTED was not available, AVAILABLE_BLOCK_SIZE will be the number of bytes of the largest available block in system memory.

BEGINNING_ADDRESS: Type Word. This parameter has three possible values. If memory is successfully allocated, BEGINNING_ADDRESS will be the memory address of the beginning of the allocated block. If sufficient memory is not available, BEGINNING_ADDRESS will be the memory address of the beginning of the largest block of memory that is available. If not even one single byte of memory is available, BEGINNING_ADDRESS will be zero.

ENDING_ADDRESS: Type Word. This parameter has two possible values. If memory allocation was successful, ENDING_ADDRESS will be the memory address of the allocated block. If there was insufficient memory for the BLOCK_SIZE_REQUESTED then ENDING_ADDRESS will be zero.

b. System Configuration Changes

If memory allocation was successful, the Operating system will have the requested block of memory reserved. If allocation was not successful, no system configuration changes will have occured.

10. Routine Testing

a. Description of Test

A simple PLZ program which calls the memory manager via allocated was written. This program outputs to the console the return code from the call to the memory manager and the other output parameters of AOLLCATE. The program was run a number of times with different input parameters. Between runs, the operating system memory status display was displayed to see the current memory allocation. DEALLOCATE was tested concurrently.

b. Results of Test

When the request was valid, ALLOCATE successfully conveyed the requests to the memory manager; memory was allocated. When unsatisfiable requests were made, ALLOCATE received and correctly interperted the responses from the memory manager. Conclusion: ALLOCATE works.

11. Reference to Listing

The program listing for ALLOCATE is on pages 324 – 325 in Appendix B.

1. Name: **DEALLOCATE**

2. Part of Utility Module

3. Written in Z-80 Assembly Language; 38 bytes.

4. Synopsis of Routine

DEALLOCATE is an assembly language routine which permits a PLZ program access to the operating system memory manager for deallocation of specific blocks of memory. DEALLOCATE has four major sections of code:

a. AREC save
b. Call of Memory Manager
c. Output Parameter setup
d. Stack clean up and return to calling routine.

5. Routine Relationship Diagram

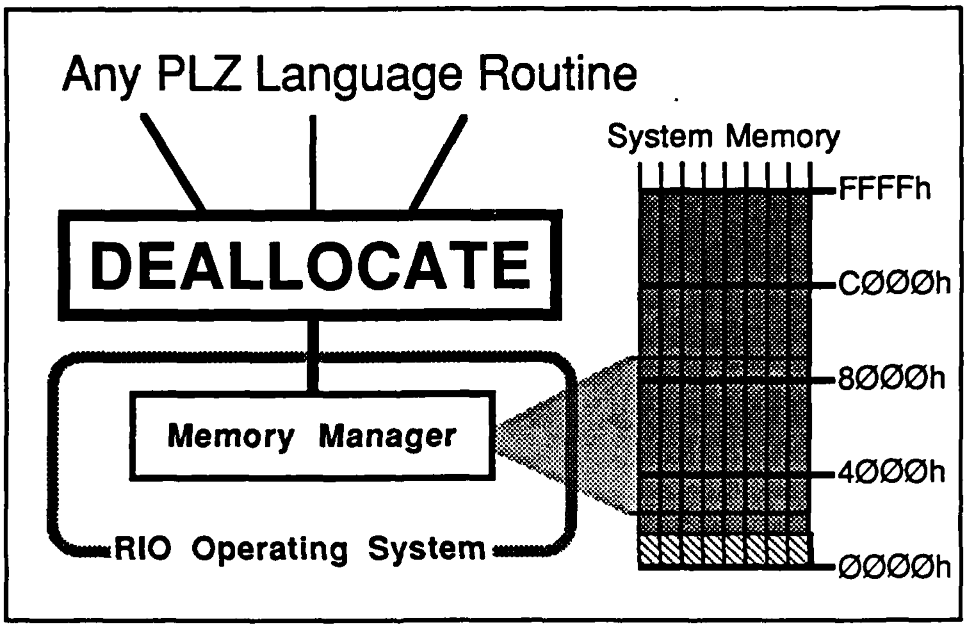


Figure 41. Relations of DEALOCATE to Calling PLZ Routines and to RIO Operating System.

6. Invocation

a. Invocation Statement

DEALLOCATE is invoked in a PLZ routine via:

RETURN_CODE := DEALLOCATE(BLOCK_SIZE, BEGINNING_ADDRESS)

where RETURN_CODE is of type Byte and BLOCK_SIZE and BEGINNING_ADDRESS are of type Word. The Utility Module must be linked in will the calling program.

b. Input Parameter Passing Schema

DEALLOCATE has two input parameters, BLOCK_SIZE and BEGINNING_ ADDRESS. BLOCK_SIZE is the number of memory locations to be deallocated. BEGINNING_ADDRESS is the address of the first memory location of the block to be deallocated. The block to be deallocated must be fully allocated at the onset of this routine.

c. Routines Which Call

DEALLOCATE can be called by any PLZ program linked with the Utility Module. Though it is not used in any of the current data collection software. An improved SIZE_DATA_BUFFER that uses ALLOCATE would force the use of DEALLOCATE near the end of module execution to free up memory.

7. Variables and Constants

a. Global

There are no true global variables or constants.

b. Constants Internal to the Module

ZERO:	ØØ hex
DEALCT_MEMORY:	Ø1 Hex, the code for deallocation of memory passed to the memory manager.
MEMORY_MANAGER:	14Ø9 Hex, the address of the memory manager entry point.

c. Internal to the Routine

Besides the CPU registers used to hold parameters (see below), DEALLOCATE has no internal variables or constants.

8. Other Routines Called

DEALLOCATE uses the system Memory Manager routine. CPU registers are used to pass parameters between DEALLOCATE and the Memory Manager. When DEALLOCATE calles memory manager:

BC holds the BLOCK_SIZE to be deallocated;
HL holds the BEGINNING_ADDRESS of the block;
A holds the request code for memory deallocation, Ø1 hex.

The Memory Manager returns to DEALLOCATE register A holding the RETURN_CODE, 8Ø hex for successful deallocation or 43 hex for memory protect violation. Memory protect violation occurs when the block identified for deallocation is not completely and continuously allocated. Note that these register stored values are the input and output parameters of DEALLOCATE.

9. Output of Routine

a. Output Parameter Passing Schema

DEALLOCATE returns a single parameter, RETURN_CODE, which indicates whether the deallocation was successful. If the deallocation was successful, RETURN_ CODE will have a value of 8Ø hex. If the deallocation is unsuccessful the RETURN_ CODE will have a value of 43 hex.

b. System Configuration Changes

If memory deallocation was successful, the block of memory specified by the input parameters will no longer be allocated. If deallocation was unsuccessful, no configuration changes will have occured.

10. Routine Testing

a. Description of Test

DEALLOCATE was tested in conjunction with ALLOCATE through a simple PLZ routine. This routine used ALLOCATE and DEALLOCATE to alter the system memory allocation. In between calls, the status of the system memory was checked via an operating system utility.

b. Results of Test

Whenever the deallocation request was valid, DEALLOCATE successfully deallocated the specified block of memory and returned the successful operation return code. When invalid deallocation requests were made, DEALLOCATE was unable to deallocate memory (as it shouldn't) and returned the proper return code for memory protect violation. Conclusion: DEALLOCATE works.

11. Reference to Listing

DEALLOCATE's listing can be found on pages 326 – 327 in Appendix B.

This page is intentionally blank.

IV. Sampler Module

Introduction to Sampler Module

The Sampler Module is a collection of twelve assembly language routines which implement a real-time clock paced data collection system. The module uses periodic interrupts from the CTC (counter/timer chip) of the MCB Board to initiate analog to digital conversions by the AIO (analog input output) board. When each conversion is complete, the digital data is read from the AIO board and placed in a buffer. The process continues until a specified number of samples has been input and stored in the buffer. This interrupt / convert / store process is preceeded by a series of initilization steps and is followed by a set of shut down and deallocation routines.

In the following paragraphs, the organization, program flow, interrupt routine selection, invocation, language, call overhead, testing, and known problems of Sampler Module are discussed. Following these discussions are the detailed descriptions of the twelve routines of the module.

Organization and Function of Sampler Module Routines

Sampler Module is organized into an executive rouine, nine subordinate routines, and two interrupt service routines. Sampler Module could have been written as a single sequence of assembly code plus the two interrupt service routines. This approach was rejected in favor of the executive / subordinate organization for three reasons. First, the executive/subordinate structure is far more readable and maintainable than a long single string of code. The executive clearly shows the high level program flow and all the module control branching; this detail would have been obscured in a large single string of code. Second, a number of the subordinate routines are complete functions developed originally in PLZ (AIO.PLZ.S Module) or used elsewhere; these routines were already functionally separate routines. Third, the functions needed in the module logically follow a building block organization, particularly the interrupt service routines. For these reasons, Sampler Module is organized into an executive routine, nine subordinate routines, and two interrupt service routines.

The twelve routines of Sampler Module and and a description of their their functions fpllows.

Routine Name	Function of Routine
SAMPLER	Executive routine of Samper Module. Calls routines VALIDATE through DEALLOCATE in turn.
VALIDATE	Verifies the correctness of the module input parameters.
ATODINIT	Initializes the AIO Board by putting the board Into polled mode and clears the analog to digital input registers.
CTC_PROGRAM	Initializes the CTC timer chip by loading the desired prescaller for the timing count and the interrupt vector.
INT_SET_UP	Establishes the parameters for the interrupt service routine including selection of TO_SAMPLE or TC_SAMPLE for the interrupt service routine.
INIT_COLLECTOR	Loads control parameters in to the CPU registers.
USER_READY?	Querries the user via the system console and keyboard for a signal to begin data collection.
START_TIMER	Loads the CTC timer with the selected time constant which completes its programming and initiates the real time clock.
COLLECTER	Loops, polling the AIO board status register and reads In converted data when an analog to digital conversion is complete. Counts the collections and ends, exiting loop, when last sample has been read.
CTC_OFF	Deactivates the interrupts and timing of the CTC.
DEALLOCATE	Loads the output parameters and deallocates stack space of the input parameters.
TO_SAMPLER	Interrupt service routine for sample periods of Ø.Ø1 seconds or less. No counter is used. Initiates an analog to digital conversion each time called.
TC_SAMPLER	Interrupt service routine for sample periods greater than Ø.Ø1 seconds. Decrements a counter each time called. When counter reaches zero, initiates an analog to digital conversion and resets the counter.

Execution Flow within Sampler Module

The flow of program execution between the executive routine SAMPLER and its nine subordinate routines is shown by Figure 42 below. SAMPLER calls its nine subordinate routines in succession with two possible branches. These branchs occur within SAMPLER and are based on the output (state of the CPU zero flag) of subordinate routines VALIDATE and USER_READY?. In both cases the branching is to abort the execution of the remaining module steps. From VALIDATE, Sampler Module execution is aborted if the input parameters supplied by the calling PLZ program are invalid. From USER_READY? execution is aborted if the User signals to abort data collection. Abortion of execution from USER_READY? requires a call to CTC_OFF to disable CTC timing and interruptions. DEALLOCATE is called from both execution abortion paths to prepare for the return to the calling PLZ routine. For more information on the internal execution and interfaces of the Sampler Module routines, please consult the detailed routine descriptions.

The interrupt service routine, either TO_SAMPLE and TC_SAMPLE, is not called by SAMPLER. Instead, the interrupt service routine executes out of routine COLLECTER. INT_SET_UP selects which interrupt service routine will be used and loads the address of the selected routine into the interrupt vector location. When a CTC issued interrupt occurs, program execution jumps to the selected interrupt service routine. When interrupts are not being serviced, the code of COLLECTER is being executed. The logic states of COLLECTER, including the jumps to the interrupt service routine, are shown in Figure 43 below. COLLECTER primarly sits in READY? checking whether an analog to digital conversion has been completed and data is ready. It is during this READY? state that interrupts will occur. The interrupt service routine, either TO_SAMPLE or TC_SAMPLE, initiates the analog to digital conversion. When data is ready from the AIO board, COLLECTER shifts to the DATA_READY state. There, COLLECTER reads in and stores the data. COLLECTER then checks to see how many samples have been read in. If there are more samples to be collected, execution shifts back to state READY?. If all the samples have been collected, execution shifts to the FINISHED state. FINISHED corrects all pointers and returns program execution to SAMPLER.

Interrupt Routine Selection

Which interrupt service routine is used depends upon the sampling period required. The CTC timer alone can generate periodic interrupts every 6.515 microseconds to 26.58 miliseconds (Ref 7: Sec 3.7). The interval between the interrupts is determined by the prescale factor (16 or 256) and the time constant given to the CTC during programming. For sampling periods within the above range, the interrupt service routine simply writes to the AIO channel select register each time an interrupt occurs. This is the procedure used by TO_SAMPLER, the "TO" standing for "Timer Only."

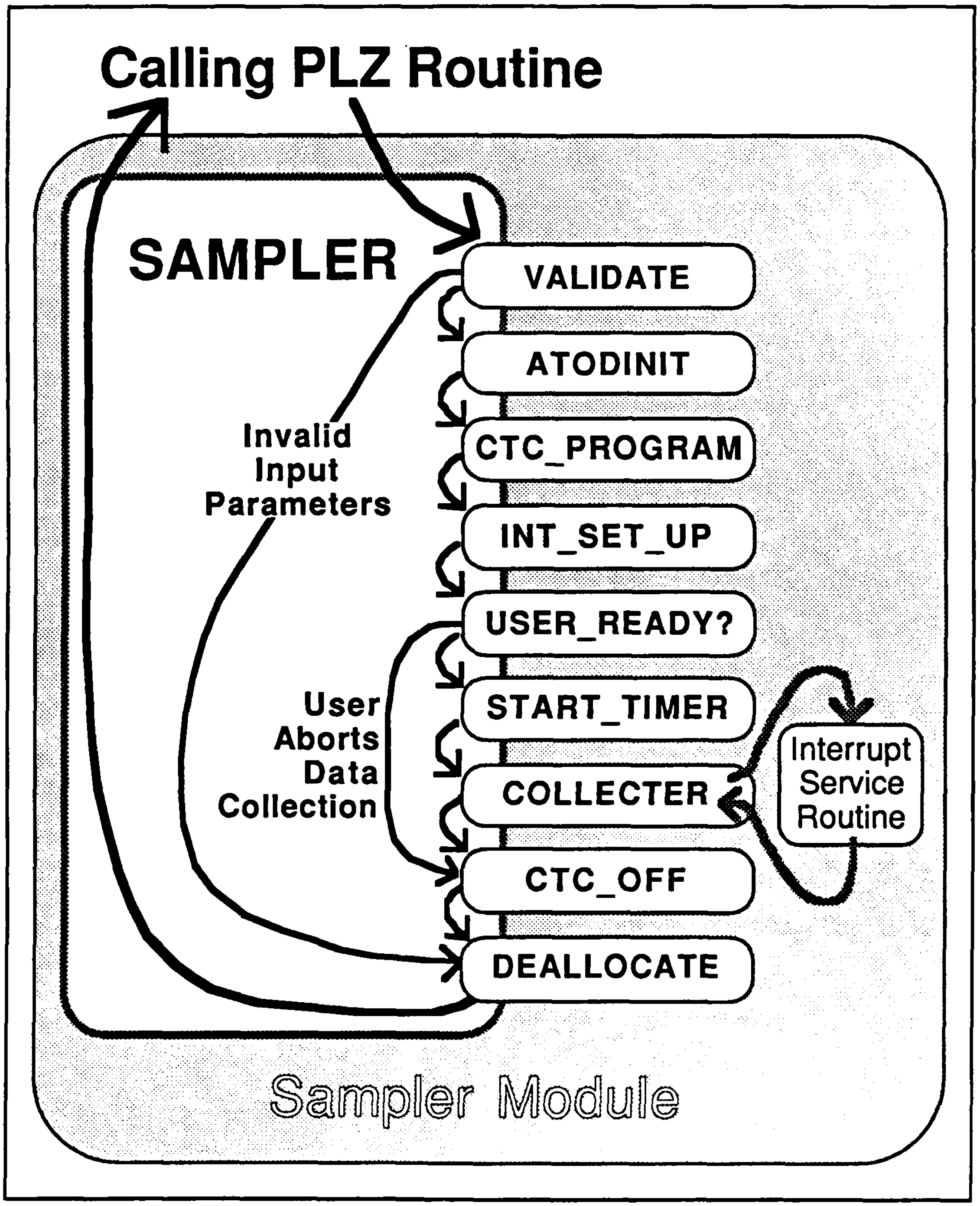


Figure 42. Relationship of SAMPLER and its Subordir ɜ'ə Routines, the Interrupt Service Routine, and to the Calling Routine.

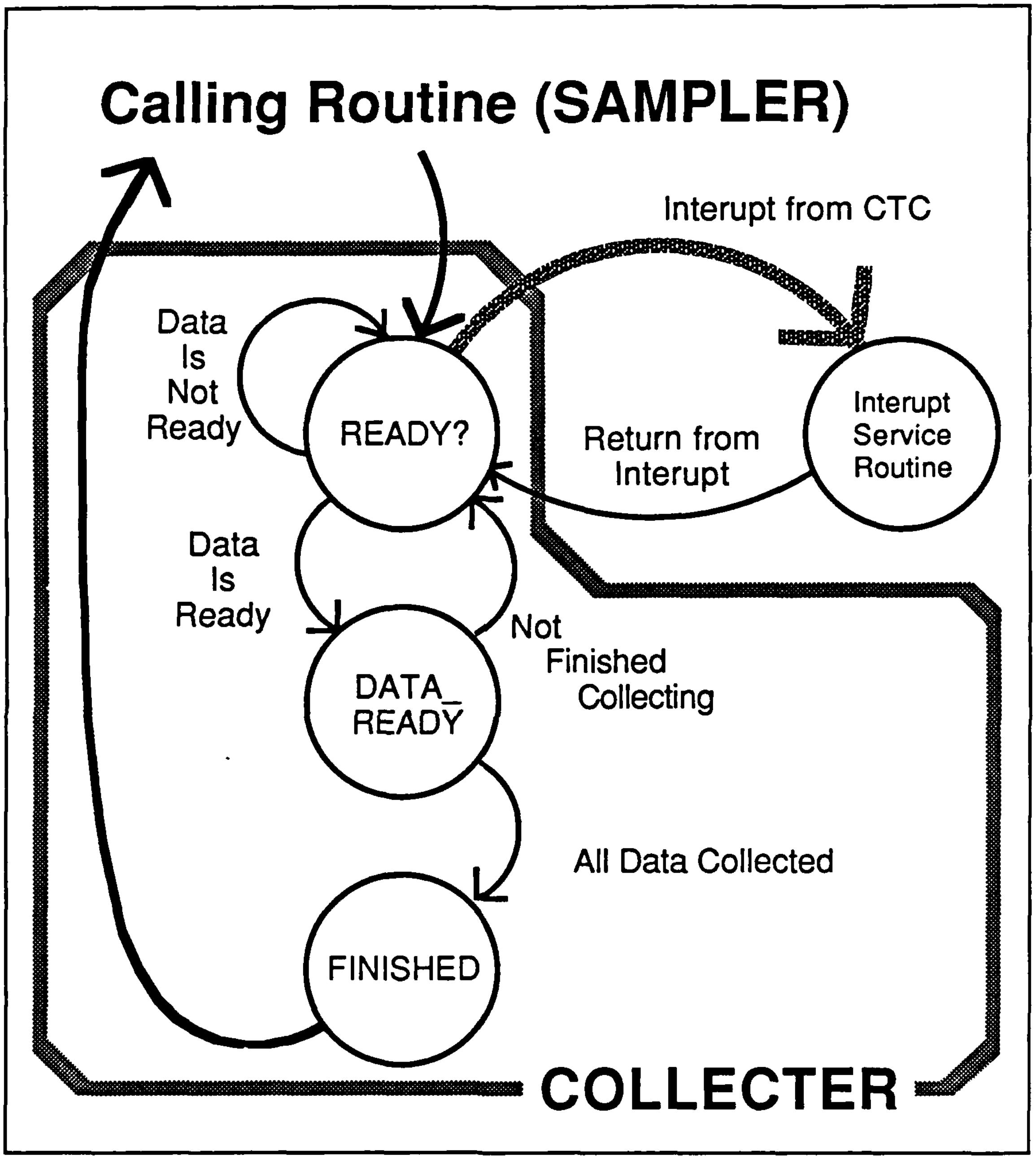


Figure 43. Operation States During Subordinate Routine COLLECTER Including the Interrupt Service Routine.

To obtain longer sampling periods, a counter must be added to the interrupt service routine. Each time the CTC issues an interrupt, the interrupt service routine decrements a counter. When the counter reaches zero, the service routine writes to the AIO channel select register and resets the counter.

For this method of generating sampling periods, three parameters are required, the CTC prescale factor, the CTC time constant, and the counter value. This method is used by TC_SAMPLER, where "TC" stands for "Timer & Counter." Given a sixteen bit counter in addition to the CTC timer, sampling periods of 1.688 miliseconds to 29.3 minutes are possible with the timer and counter combination.

Figure 44 below shows the sampling period ranges of the various combinations of CTC timers and sixteen bit counters. Slow Timer refers to a CTC timer using a prescale factor of 256. Fast Timer refers to a CTC timer using a prescale factor of 16. As shown in the figure, the sampling period ranges of the timers and the timer/counter combinations overlap.

For this thesis effort, aribritrary break points to choose between the four different timing methods were selected. For sample periods below Ø.Ø1 seconds, the CTC timer only (interrupt service routine TO_SAMPLE) is used. For periods less than Ø.ØØ1 seconds a prescale factor of 16 is loaded into the CTC; for periods Ø.ØØ1 seconds to Ø.Ø1 seconds , the prescale facter is 256. The timer counter combination (interrupt service routine TC_SAMPLE) is used for sampling periods Ø.Ø1 seconds and above. For periods from Ø.Ø1 seconds up to 1.Ø second, a fast timer (prescale factor of 16) is used with the 16 bit counter. For periods from 1.Ø second to the maximum time possible of 29.3 minutes, the slow timer (prescale of 256) is used. The shaded areas on Figure 44 show the employed ranges for each timer/counter combination.

The parameters which program the CTC and the sixteen bit timer are input parameters to Sampler Module. The calling PLZ routine establishes these values based on the user's desired sampling period and the routine break points discussed above. Routine INT_SET_UP looks at the input parameter COUNT, the sixteen bit down counter value. If COUNT is zero, INT_SET_UP selectes TO_SAMPLE as the interrupt service routine. If COUNT is nonzero, TC_SAMPLE is used. Please note that the calling PLZ routine does not use the full range of the fast timer only combination. To allow sufficient time for the analog to digital conversion to take place, the shortest sampling period actually employed is 5Ø.Ø microseconds.

Invocation of Sampler Module

As shown by Figure 42 above, Sampler Module is called from a PLZ program. The PLZ program supplies the three values needed to program the real time clock , specifies how many samples are to be collected, and names the analog input channel is to be used. The executive routine SAMPLER is the program interface between the calling PLZ routine and all of Sampler Module.

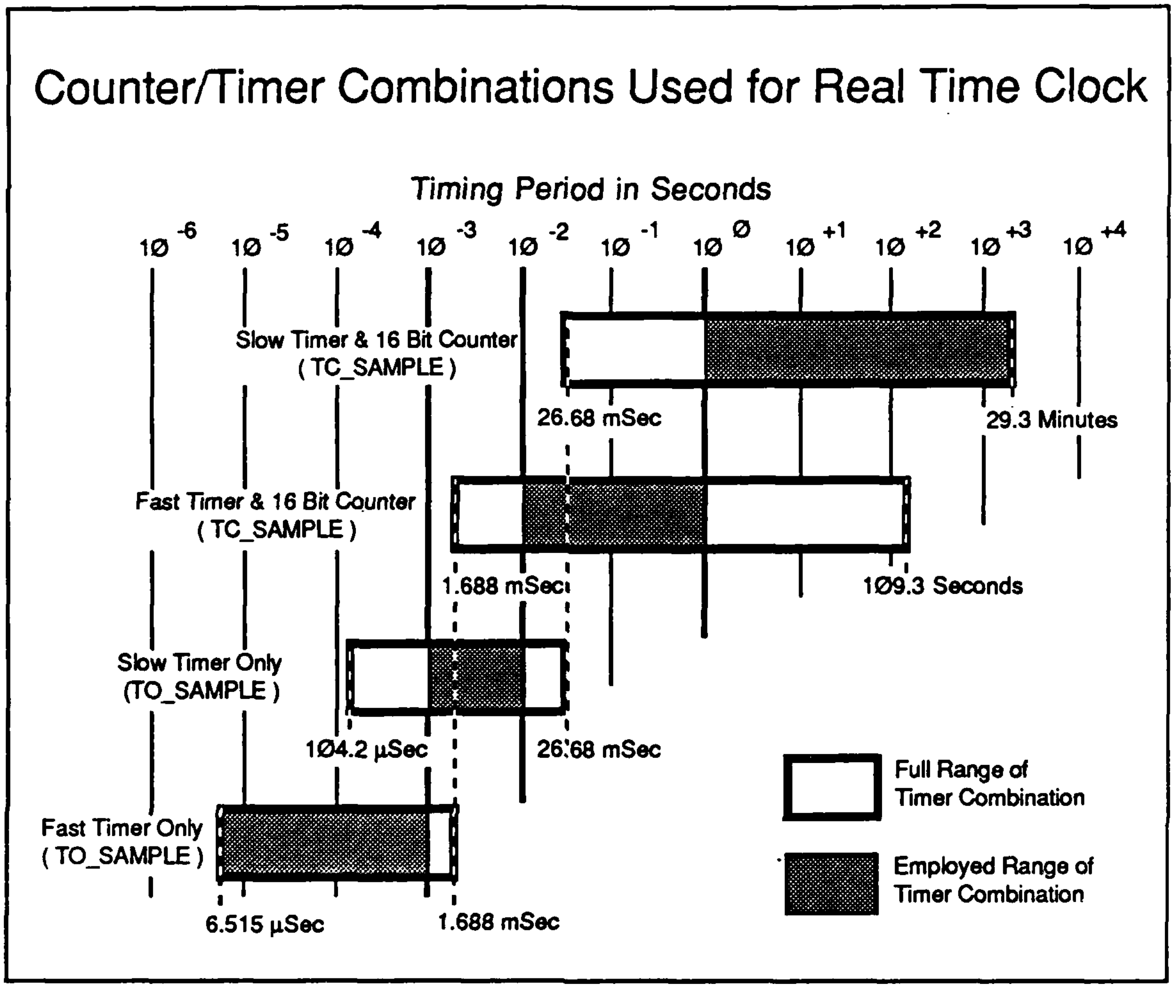


Figure 44. Counter/Timer Combinations Used for Real Time Clock

SAMPLER, and hence all of Sampler Module, is invoked from a PLZ routine with

```
ERROR_CODE, LAST_DATA :=
                    SAMPLER(  IO_CHANNEL, CTC_MODE,
                              TIME_CNST, COUNT,
                              NUM_SAMPLES, FIRST_DATA  )
```

The purpose and type of the input and output parameters is:

Parameter Name	Type	Parameter Purpose
IO_CHANNEL	Byte	Selects which one of the 16 possible AIO board analog input channels is to be used.

Parameter Name	Type	Parameter Purpose
CTC_MODE	Byte	Passes the first half of the command used to program the CTC to issue interrupts at the desired rate.
TIME_CNST	Byte	Passes the second half of the CTC programming command.
COUNT	Word	The number of CTC interrupts required between data collections. This parameter is used only for long timer periods.
NUM_SAMPLES	Word	The number of data samples to be read in.
FIRST_DATA	Pointer-to-Byte	A pointer to the first memory location for the stroage of the data read in.
LAST_DATA	Pointer-to-Byte	Outputs the pointer to the last memory location that data was stored in.
ERROR_CODE	Byte	Passes back to the calling routine an error message if the calling routine's inputs were improper.

Although the executive routine SAMPLER is the sole program execution interface to the calling PLZ routine, SAMPLER does not use any of the subroutine call parameters. Instead, the input and output parameters are employed only by the subordinate routines which need them. From the calling PLZ routine's perspective, Sampler Module is simply a single subroutine; the executive/subordinate organization of these assembly language routines is neither visible nor important.

Selection of Assembly Language for Sampler Module

The routines of Sampler Module were written in assembly language primarily to gain a speed of execution advantage. Given the access to the system provided by the Utility Module routines, Sampler Module could have been written in PLZ. In fact, some of the PLZ language routines of the AIO.PLZ.S Module are precursors of some of the assembly language routines in Sampler Module. The only problem with PLZ is speed. The overhead required by a PLZ routine would have precluded the shorter sampling periods achieved by using assembly language routines. With PLZ and the Utility Module routines, the polling of the AIO status register would have required a PLZ call to IOIN, execution of IOIN (11 instructions), and the return to the PLZ routine. This sequence would have required approximately 2ØØ microseconds to execute (see Appendix C). With assembly language the whole loop is just four instructions requiring about 16 microseconds to execute. Another example is the calculation of the CTC timer and sixteen bit counter values for the sampling period. These could have been done in assembly language with the addition of some math utilities. However, in PLZ the math and high level logical branching instructions were already present. By having the assembly language Sampler Module interface with a PLZ parent routine the best of both worlds was obtained, the speed and direct hardware access of assembly language coupled with the higher level programming of PLZ.

Overhead for PLZ Subroutine Call of Sampler Module

The overhead for an assembly language routine to be called by a PLZ routine was extensively discussed in the introduction to the Utility Module. Rather than repeate that discussion here, please refer to the Utility Module discussion and sample AREC for more information of PLZ parameter passing schema. The figure below shows the PLZ Activation Record (AREC) for the parent routine's call of Sampler Module.

External Calls of the Sampler Module

The routines of Sampler Module use no other subroutines. However the RIO Operating System and several hardware elements of the MCB development system are called. The items called, the calling routine, and the purpose of the calls are fully detailed in the routine descriptions.

Testing of Sampler Module

Three types of tests were performrned on the routines of Sampler Module. First, one of the routines, ATODINIT, was individually tested. Second, portions of Sampler Module were tested using the RIO debugger. Third, a short PLZ module was written solely to call and test Sampler Module. ATODINIT was an established routine which functioned properly. Its individual testing was its prior use. The rest of the testing was far more involved.

The testing with the debugging routine was limited in application and

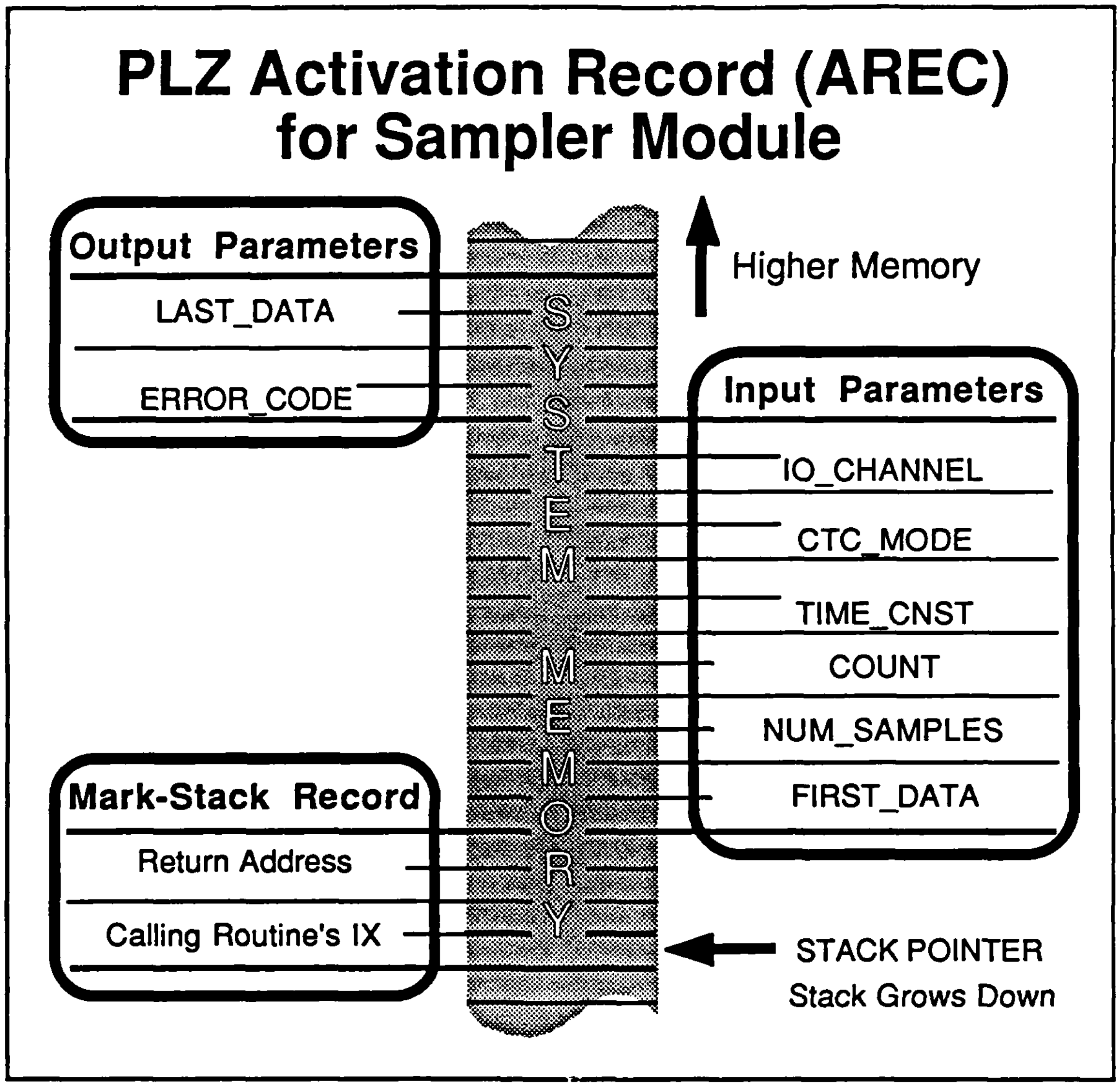


Figure 45. Activation Record for Call of Sampler Module.

somewhat cumbersome to accomplish. The debugging routine is interrupt driven; Sampler Module is interrupt driven. Thus, the debugger could not be readily used to test the interrupting portion of Sampler Module. The debugger was used in conjunction with a logic analyzer to examine the Sampler Module routines which set the Z-8Ø registers and worked with the AIO board. The CTC related routines which delt with interrupts were not tested with the debugger. The logic analyzer was used to trap the input/output port calls. One of the more difficult actions was to manually insert the parameters that a calling PLZ would normaly have placed in the system stack. This action was aided by the symbolic capabilitites of debugger which allowed access by name rather than hexidecimal addressses. The debugger testing showed that the tested protions were functioning properly. The AIO board was receiving the proper commands and information could be obtained from it.

Things didn't go as well with the PLZ routine testing. For this test, a short PLZ routine was written for the sole purpose of calling Sampler Module. The routine consisted of the necessary variable definitions, a call of SAMPLER, and screen output of the return parameters. Post-test, system memory was then examined with the ROM monitor routine to see that data had been loaded into memory. During the test a slowly varying square wave was fed into the analog input. A square wave was used so that only two digital values should appear in the memory. Well, the program executed, Sampler Module requested a go signal, interrupts began, data was collected in memory. However, program execution never left Sampler Module to return to the PLZ routine. A whole bunch of time was spent trying to find out why this occured. No answer was found.

Known Problems in Sampler Module

As discussed in the testing section above, Sampler Module never properly interfaced with a calling PLZ routine. The cause of this problem is still unknown.

Content of Detailed Routine Descriptions

Following are detailed descriptions of the twelve assembly language routines of the Sampler Module. With a few exceptions, the following items will be presented for each of the routines.

1. Routine Name
2. Module Name and Role of Routine
3. Language and Length of Routine
4. Synopsis of Routine
5. Routine Relationship Diagram
6. Invocation of Routine
7. Variables and Constants Used by Routine
8. Discussion of Other Routines Called

9. Output of Routine

10. Routine Testing

11. Reference to Routine Listing

The routine testing discussions are limited to activities beyond those addressed in the module testing discussion above. The listing of routines of Sampler Module are in Appendix D.

1. Routine Name: **SAMPLER**

2. Executive Routine of Sampler Module

3. Written in Z-8Ø assembly language; 16 lines (42 bytes) of code.

4. Synopsis of Routine

SAMPLER is the executive routine of Sampler Module. This assembly language routine is the entry routine of the module and is in effect the routine called by the PLZ program. It manages overall program flow within the module by calling nine subordinate routines an by using conditional branching based on error checking and user readiness checks. SAMPLER also handles a portion of the PLZ subroutine call overhead and performs the jump back to the calling PLZ routine. Figure 42, in the introduction to Sampler Module, shows the flow of SAMLER, the conditional branches, and the routines called by SAMPLER.

The following discussions are specifically restricted to the 16 lines of code which are called SAMPLER. This is a rather arbitriary distinction. While SAMPLER does little more than call nine other routines, without SAMPLER those routines would not function. It is perhaps best to view SAMPLER as an organizer of the Sampler Module rather than a complete software routine. The discussion that follows centers on this organizer function.

5. Invocation

Since the first line of SAMPLER is the entry point for the Sampler Module, SAMPLER is the routine called by the parent PLZ program. Thus, the invocation of SAMPLER is the same as the innvocation for the Sampler Module discussed previously. However, SAMPLER itself uses none of the input and output parameters of that invocation; these parameters are used by the subordinate routines in the Sampler Module. The subordinate routines do depend upon SAMPLER to load the IX register with the stack pointer value so they can reach the parameters with offsets.

6. Variables and Constants

SAMPLER uses no declaired variables or constants. It does place the current value of the stack pointer into the IX register so that it's subordinate routines can access the input and output parameters with offsets from the IX value. SAMPLER also uses the Zero Flag of the Z-8Ø CPU to determine whether to branch upon the completion of VALIDATE and USER_READY?

7. Other Routines Called

As discussed in the introduction to the Sampler Module, SAMPLER is the executive routine for the module. As such, all other routines of the module are called, either directly or indirectly, by SAMPLER. The names and functions of these routines was also presented in the module introduction. Figure 42 in the introduction to Sampler Module shows when in the flow of SAMPLER each subordinate routine is called.

There are no true parameters passed between SAMPLER and its subordinate routines. The only communication SAMPLER uses is the status of the zero flag upon completion of VALIDATE and USER_READY?. For both of these routines, a nonzero flag tells SAMPLER to abort. From VALIDATE, SAMPLER just jumps to DEALLOCATE to satisfy PLZ subroutine termination requirements; from USER_READY? SAMPLER must call both CTC_OFF (to clear the counter timer chip) and DEALLOCATE. SAMPLER expects DEALLOCATE to load the HL register with return address of the calling routine.

8. Output of Routine

As stated above, SAMPLER, the entry routine of the Sampler Module, does not pass parameters. The output parameters for the module are loaded by VALIDATE, USER_READY?, or COLLECTER. Similarly, SAMPLER by itself does not cause any system configuration changes, though the unaborted execution of the Sampler Module will result in a number of analog to digital conversion and storage of those conversions in system memory.

9. Routine Testing

SAMPLER was not independently tested.

10. Reference to Listing

The program listing of SAMPER is on page 318 in Appendix D.

1. Routine Name: **VALIDATE**

2. Subordinate Routine of Sampler Module

3. Written in Z-8Ø assembly language; 14 lines, 3Ø bytes, of code.

4. Synopsis of Routine

VALIDATE is a defensive error checking routine for Sampler Module. Upon being called by SAMPLER, VALIDATE compares the input parameters against their defined ranges and values. If an out of tollerance parameter is detected, VALIDATE loads a descriptive error code into the output parameter ERROR_CODE's location and returns to SAMPLER. The Z-8Ø CPU zero flag, if reset by the comparisons, informs SAMPLER that the input parameters were not valid.

VALIDATE looks at two input parameters, IO_CHANNEL and CTC_ MODE. IO_CHANNEL has a defined range of zero to fifteen. If IO_CHANNEL has a value greater than fifteen, ERROR_CODE is set to the constant CHANNEL_ INVALID. CTC_MODE has two possible values represented by the constants FAST_MODE and SLOW_MODE. If CTC_MODE has any other value, ERROR_ CODE is set to the constant MODE_INVALID.

5. Routine Relationship Diagram

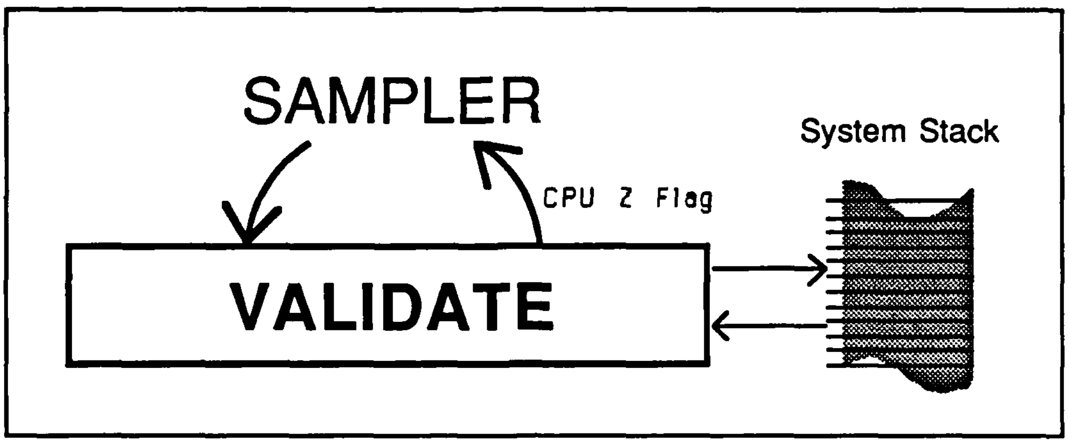


Figure 46. Relationship of VALIDATE to SAMPLER and the System Stack.

6. Invocation

VALIDATE, as an assembly language subroutine, is invoked by SAMPLER solely by its name through the Z-8Ø CALL instruction. Though VALIDATE has no formal parameter list upon invocation, it uses two of the input parameters to the Sampler Module, IO_CHANNEL and CTC_MODE, and one output parameter, ERROR_CODE. VALIDATE accesses these parameters through offsets from the IX register. This is in accordance with PLZ paramter passing procedures discussed in the introduction to the Sampler Module and in the Utility Module discussion.

VALIDATE also uses the Z-8Ø zero flag to inform SAMPLER whether the input parameter were correct. In the four comparisons are performed by VALIDATE, a nonzero result means the input parameter is out of range. The CPU's zero flag is set by the nonzero result and is not altered by the load and jump relative commands which follow the compairson. Thus, upon return to SAMPLER a true zero flag means the input parameters were correct and a false zero flag indicates flawed input.

7. Variables and Constants

a. Global

Beyond the input and output parameters IO_CHANNEL, CTC_MODE, and ERROR_CODE, VALIDATE uses no globally defined variables. The globally defined constants used by VALIDATE are:

Constant Name	Value	Definition
CHANNEL_INVALID	CA hex	Error Code for bad channel number code
FAST_MODE	87 hex	CTC command for prescale of 16
SLOW_MODE	A7 hex	CTC command for prescale of 256
MODE_INVALID	CC hex	Error Code for wrong CTC command

b. Module

VALIDATE uses no module variables beyond employing the CPU zero flag to indicate acceptable input parameters. The module level constants used by VALIDATE are

Constant Name	Value	Definition
IO_CHANNEL	ØE hex	IX register offset for the input parameter IO_CHANNEL
UPPER_FOUR	1111ØØØØ	A mask to find higher order one's.
ERROR_CODE	1Ø hex	IX offset for output parameter ERROR_CODE
CTC_MODE	ØC hex	IX offset for input parameter CTC_MODE

8. Other Routines Called

VALIDATE calls no other routines.

9. Output of Routine

a. Parameter Passing Schema

VALIDATE loads the output parameter ERROR_CODE with the appropriate code when it detects an invalid input parameter. The Z-8Ø zero flag passes back to SAMPLER whether the input parameters were valid or not.

b. System Configuration Changes

VALIDATE produces no system configuration changes.

10. Routine Testing

a. Description of Test

VALIDATE was tested in conjunction with the rest of the Sampler Module. Specifically for VALIDATE, invalid channel numbers (greater than 15) and CTC commands were passed into Sampler.

b. Results of Test

VALIDATE caught the invalid input parameters; VALIDATE did not reject valid input parameters.

11. Reference to Listing

The listing of VALIDATE is on page 339 in Appendix D.

1. Routine Name: **ATODINIT**

2. Subordinate Routine of Sampler Module

3. Written in Z-8Ø assembly language; 13 lines (21 bytes) of code.

4. Synopsis of Routine

ATODINIT initializes the analog to digital converter of the AIO board. This assembly language routine is based on the PLZ language routine AIO_INIT. Upon being called by SAMPLER, ATODINIT performs five operations as shown in the figure below. First the AF registers are saved and the Z-8Ø interrupts are disabled. The AF register save is an artifact of the routine's use in booting the development system. The interrupts are disable to prevent inadvertant interrupts from the AIO board during its programming. Next, ATODINIT sets the two AIO ports to input mode by writing the command INMODE to both ports' command registers. Third, the AIO is placed in polled mode by writing the command INT-DISABLE to the command registers. Fourth, the data registers (upper and lower) are cleard to ready the board for input. Last, the Z-8Ø interrupts are enabled, the AF register values restored, and control is returned to SAMPLER.

5. Routine Relationship Diagram

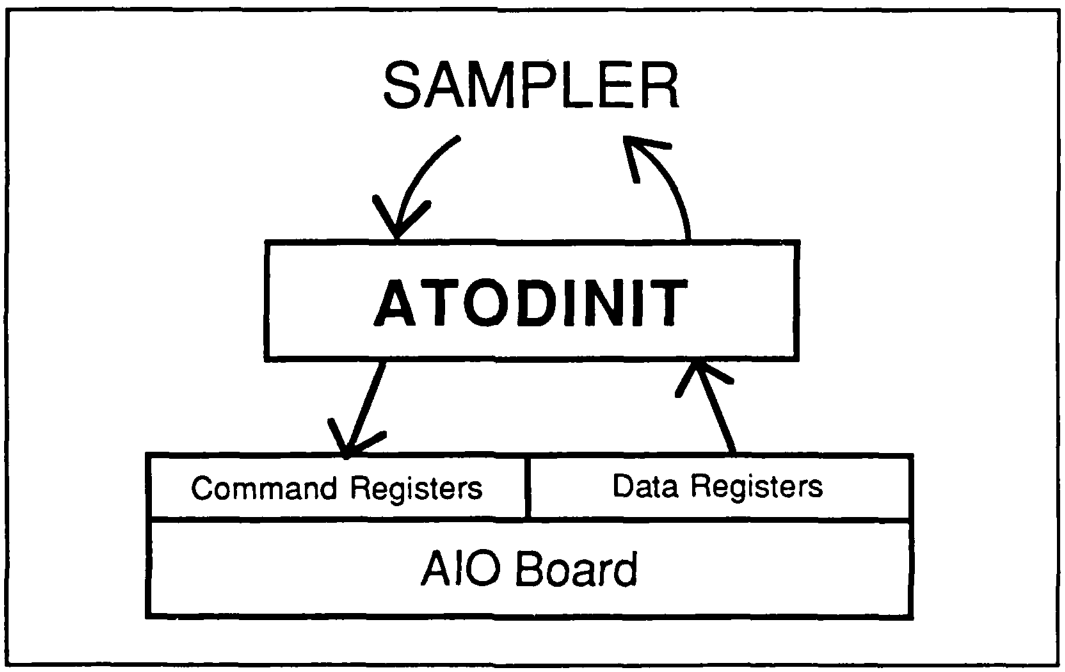
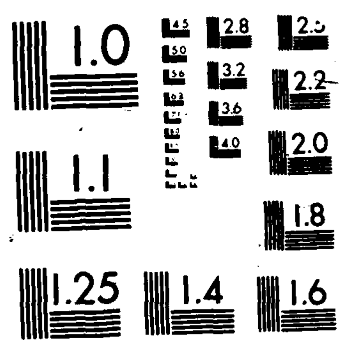


Figure 47. Relationship of ATODINIT to SAMPLER and AIO Board.

AD-A172 823 DESIGN AND PARTIAL IMPLEMENTATION OF A COMPUTER CONTROLLED DATA COLLECTION SYSTEM(U) AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH SCHOOL OF ENGI.. L E LUTZ 3/5

UNCLASSIFIED FEB 86 AFIT/GE/ENG/86M-1 F/G 9/2 NL


1.0
2.8
3.2
3.6
4.0
2.2
2.0
1.1
1.8
1.25
1.4
1.6

6. Invocation

ATODINIT is invoked simply by name. It is self contained, having no input or output parameters.

7. Variables and Constants

ATODINIT uses no variables. It uses six global constants for commands and IO port addresses. Their names, values, and definitions are

Constant Name	Value	Definition
InMode	4F hex	AIO Command for Polled AtoD Conversions
CMD_A_PORT	22 hex	Address of AIO Port A Command Register
CMD_B_PORT	23 hex	Address of AIO Port B Command Register
INTDisable	Ø7 hex	AIO Command for Disabled Interrupts
DataLower	2Ø hex	Address of AIO Lower Data Register
DataUpper	21 hex	Address of AIO Upper Data Register

8. Other Routines Called

ATODINIT calls no other routines. It does write commands to the AIO Board.

9. Output of Routine

ATODINIT has no outputs. Its impact upon system configuration is that the AIO board in now in polled input mode.

10. Routine Testing

ATODINIT was not individually tested. ATODINIT is based on AIO_INIT and is used in other programs where it functions properly.

11. Reference to Listing

The listing of ATODINIT's assembly language code is on page 340 in Appendix D.

1. Routine Name: **CTC_PROGRAM**

2. Subordinate Routine of Sampler Module

3. Written in Z-8Ø assembly language; 5 lines (1Ø bytes) of code.

4. Synopsis of Routine

CTC_PROGRAM performs the initial two thirds of Counter Timer Chip One (CTC1) programming by writing the timer mode command and the CTC portion of the interrupt vector to the Channel Ø Command Register. CTC_PROGRAM obtains the mode command from the system stack as it is the Sampler Module input parameter CTC_MODE. The remaining one third of the CTC programming is accomplished by START_TIMER.

5. Routine Relationship Diagram

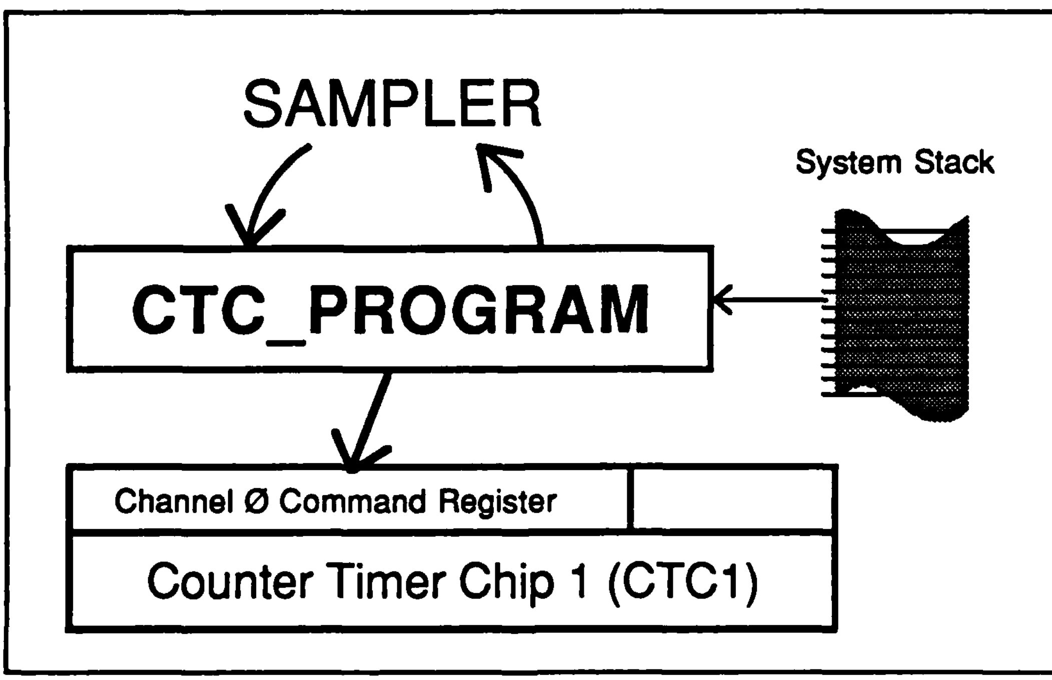


Figure 48. Relationship of CTC_PROGRAM to SAMPLER, the CTC1, and the System Stack.

6. Invocation

As an assembly language subroutine, CTC_PROGRAM is invoked by name only with the instruction CALL CTC_PROGRAM. There are no parameters formally passed.

7. Variables and Constants

CTC_PROGRAM uses one variable, the input parameter CTC_MODE, which it obtains from the system stack using module constant CTC_MODE. CTC_MODE (the variable) has two possible values 87 hex and A7 hex for fast timer with interrupts and slow timer with interrupts respectively. The fast timer uses a prescale factor of 16; the slow timer uses a prescale factor of 256. The calling PLZ routine selects which command is to be used and loads CTC_MODE appropriately.

CTC_PROGRAM uses three module constants. Their names, values, and deficitions are

Constant Name	Value	Definition
CTC_MODE	ØC hex	IX reg. offset for input parameter CTC_MODE
CTC1_CMD	84 hex	Address of CTC#1, channel Ø, command reg.
INT_VECTOR	4Ø hex	The CTC's portion of the Interrupt Vector

Note: the other half of the interrupt vector is in the Z-8Ø CPU and is a system level constant of 14 hex. The combination of the two halves yields the address 144Ø hex, the location in the interrrupt jump table where the address of the interrupt service routine will be placed by INT_SET_UP.

8. Other Routines Called

CTC_PROGRAM calls no other routines.

9. Output of Routine

CTC_PROGRAM has no output parameters. Upon completion of CTC_PROGRAM, the CTC is dormant, two thirds of the way programed to issue periodic interrupts.

10. Routine Testing

CTC_PROGRAM was not individually tested. It was tested with the rest of the Sampler Module routines.

11. Reference to Listing

The program listing of CTC_PROGRAM is on page 341 in Appendix D.

1. Routine Name: **INT_SET_UP**

2. Subordinate Routine of Sampler Module

3. Written in Z-8Ø assembly language; 9 lines (19 bytes) of code.

4. Synopsis of Routine

INT_SET_UP establishes the interrupt service routine for Sampler Module. There are two parts to this action. First, the analog input channel number is loaded into the alternate A register (A′) of the Z-8Ø CPU. INT_SET_UP gets the channel number from the input parameter IO_CHANNEL. The alternate register set is used by the interrupt service routine. Second, INT_SET_UP selects which interrupt service routine will be used based on the input parameter COUNT and loads the address of the selected routine into the interrupt jump table. If COUNT has a value of zero, routine TO_SAMPLE will be the interrupt service routine. If COUNT is nonzero, TC_SAMPLE will be used and INT_SET_UP loads the counter values into the BC′ and DE′ registers. The starting address of the selected routine is placed in memory location 144Ø hex, the interrupt jump table location for CTC1, channel Ø responses.

5. Routine Relationships Diagram

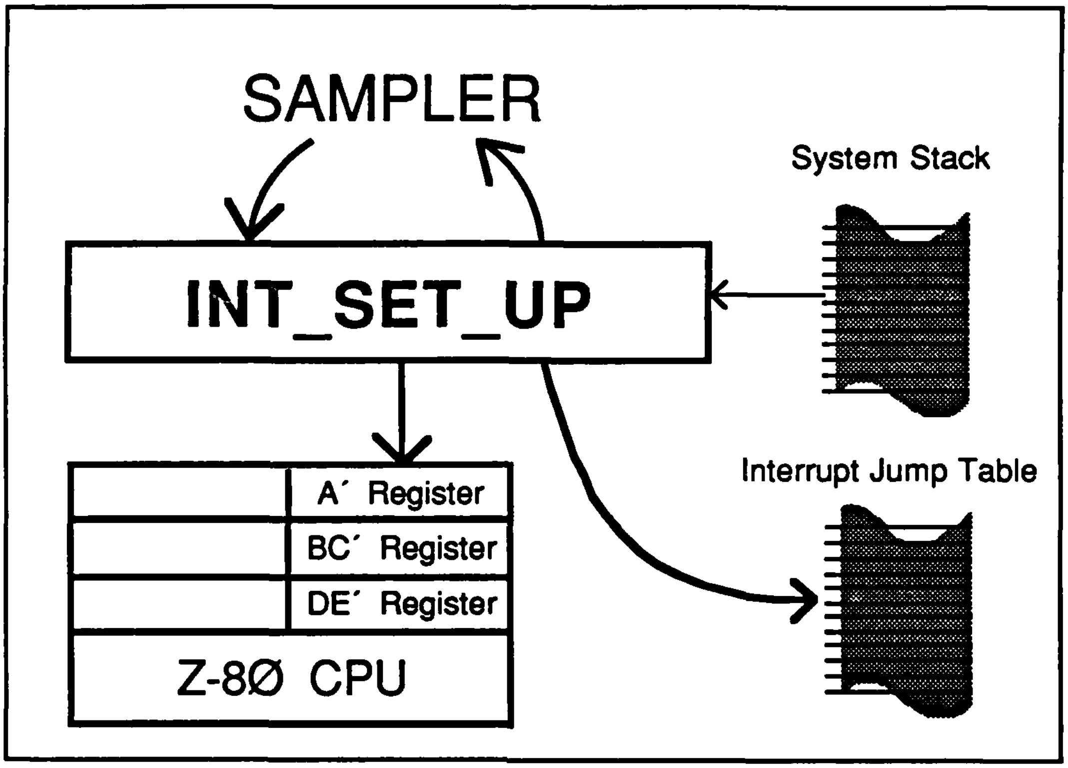


Figure 49. Relationship of INT_SET_UP to SAMPLER, the System Stack, the Interrupt Jump Table, and the Z-8Ø CPU Alternate Registers.

6. Invocation

INT_SET_UP is invoked with "CALL INT_SET_UP". Being an assembly language routine, there are no formal parameter passing lists. INT_SET_UP does expect SAMPLER to have properly set the IX register so that input parameters can be obtained via IX register offsets.

7. Variables and Constants

INT_SET_UP uses the input parameters IO_CHANNEL and COUNT. It uses six global constants for IX register offsets, interrupt service routine addresses, and the interrupt jump table address. These constants, their values, and their definitions follow.

Constant Name	Value	Definition
IO_CHANNEL	ØE hex	IX offset for input parameter IO_CHANNEL
COUNT	Ø8 hex	IX register offset for input parameter COUNT
ZERO	ØØ hex	Just zero
TO_SAMPLE	undefined	Beginning Address of Interrupt Service Routine TO_SAMPLE, defined upon program load
TC_SAMPLE	undefined	Beginning Address of Interrupt Service Routine TC_SAMPLE, defined upon program load
INT_JUMP_TABLE	144Ø hex	Address of Interrupt Jump Table location for CTC1, Channel Ø Interrupt Services

8. Other Routines Called

INT_SET_UP calls no other routines.

9. Output of Routine

INT_SET_UP has no output parameters. Its impact on system configuration is the loading of the selected interrupt service routine's starting address into the interrupt jump table and the loading of the CPU's alternate register set with the values needed by the interrupt service routine.

10. Routine Testing

INT_SET_UP was not specifically individually tested. However, during the overall testing of Samper Module, it was verified that the proper addresses were loaded into the interrrupt jump table and the CPU alternate registers were loaded with the proper values.

11. Reference to Listing

The program listing of INT_SET_UP is on pages 342-343 in Appendix D.

1. Routine Name: **INIT_COLLECTOR**

2. Subordinate Routine of Sampler Module

3. Written in Z-8Ø assembly language; 5 lines (13 bytes) of code.

4. Synopsis of Routine

INIT_COLLECTER loads into the Z-8Ø CPU's primary register set the values required by routine COLLECTER to load data into the memory buffer and to count the number of samples collected. The address for the first storage location, FIRST_DATA, is loaded into the DE register and the number of samples to be collected, NUM_SAMPLES, is loaded into the BC register. INIT_COLLECTER obtains the values from the system stack as they are input parameters to Sampler Module from the calling PLZ routine.

5. Routine Relationship Diagram

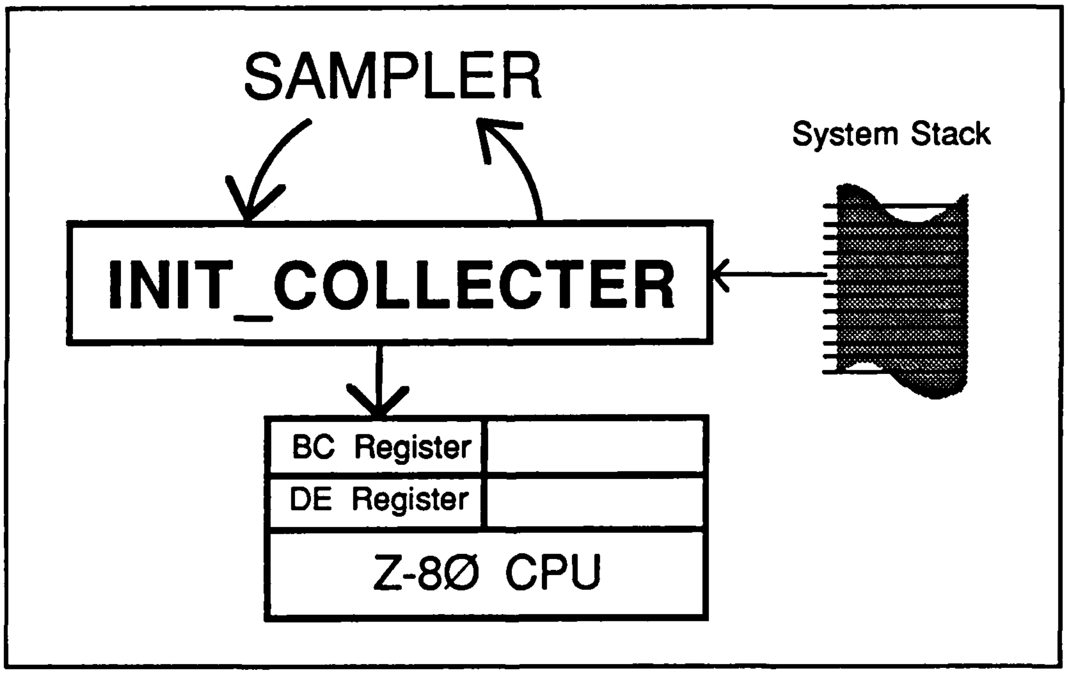


Figure 50. Relationship of INIT_COLLECTER to SAMPLER, the System Stack, and the Primay Registers of the Z-8Ø CPU.

6. Invocation

INIT_COLLECTER is called by SAMPLER though the Z-8Ø instruction CALL.

7. Variables and Constants

INIT_COLLECTER uses two input parameters, FIRST_DATA and NUM_SAMPLES, which it obtains from the system stack with two module constants. These constants, their values, and their definitions are

Constant Name	Value	Definition
FIRST_DATA	Ø4 hex	IX offset for input parameter FIRST_DATA
NUM_SAMPLES	Ø6 hex	IX offset for input parameter NUM_SAMPLES

8. Other Routines Called

INIT_COLLECTER calls no other routines.

9. Output of Routine

The sole effect of INIT_COLLECTER is the loading of the BC and DE registers with the values of the input parameters NUM_SAMPLES and FIRST_DATA.

10. Routine Testing

INIT_COLLECTER was not tested apart from the rest of the Sampler Module routines.

11. Reference to Listing

INIT_COLLECTER's program listing is on page 344 in Appendix D.

1. Routine Name: **USER_READY?**

2. Subordinate Routine of Sampler Module

3. Written in Z-8Ø assembly language; 33 lines (86 bytes) of code.

4. Synopsis of Routine

USER_READY? asks the user of the system whether all is ready for data collection. It serves as the "trigger" to begin the data collection. Figure 51 below shows USER_READY?'s relationship to SAMPLER and the operating system. For this thesis effort, the user typing a "Y" on the system keyboard tells Sampler Module to begin data collection. If other types of triggers were desired, alternatives to USER_READY? could be written and substituted into Sampler Module.

The sequence of operaions in USER_READY? is shown in Figure 52 below. USER_READY? begins by loading the output parameter ERROR_CODE with FALSE, indicating no error. Then USER_READY? calls the operating system to output the message "Collection system ready. Begin ?" to the system console. This call requires extensive preparation and loading of a transfer buffer. Next USER_READY? again calls the system to obtain the user's response from the system keyboard. This call also requires extensive preparation and loading of the transfer buffer. Execution will remain with the operating system until the user types in a character. Thus execution of Sampler Module is suspended until the user responds. When the user responds, USER_ READY? checks to see whether the character typed in is a "Y". If it is, USER_READY? exits to SAMPLER. Otherwise, ABORT is loaded into the output parameter ERROR_CODE. The failed compairson of the input character with "Y" puts the zero flag to zero. The zero flag's status will be retained during the return to SAMPLER and will indicate to SAMPLER that the user has aborted the data collection.

5. Routine Diagrams

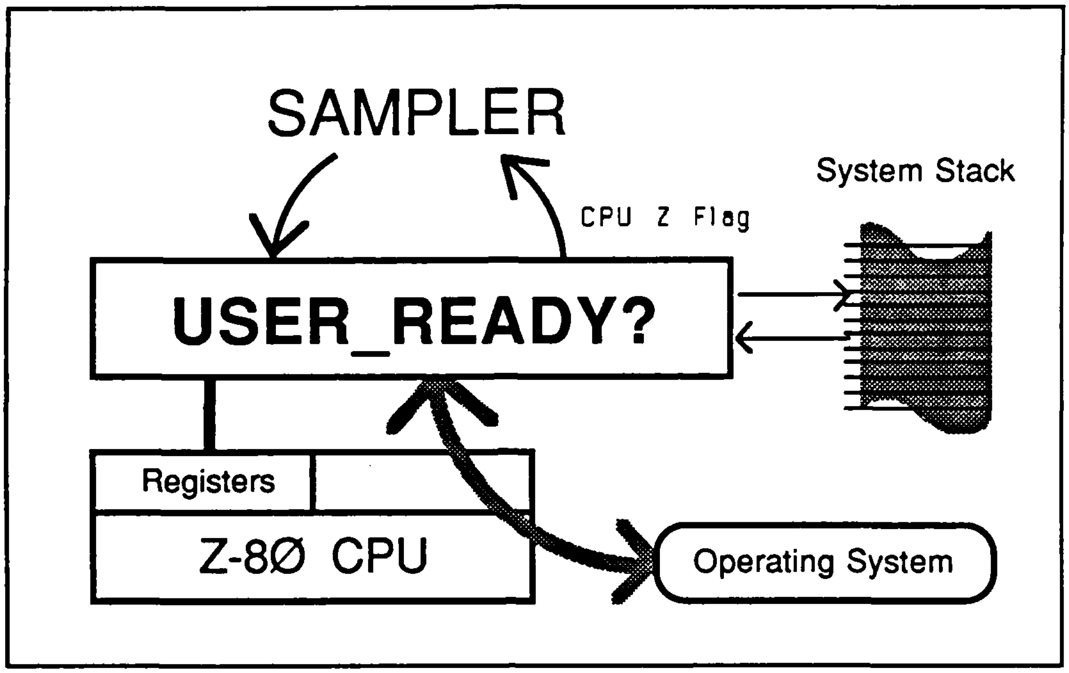


Figure 51. Relationship of USER_READY? to SAMPLER, the System Stack, the Z-8Ø Primary Registers and the RIO Operating System.

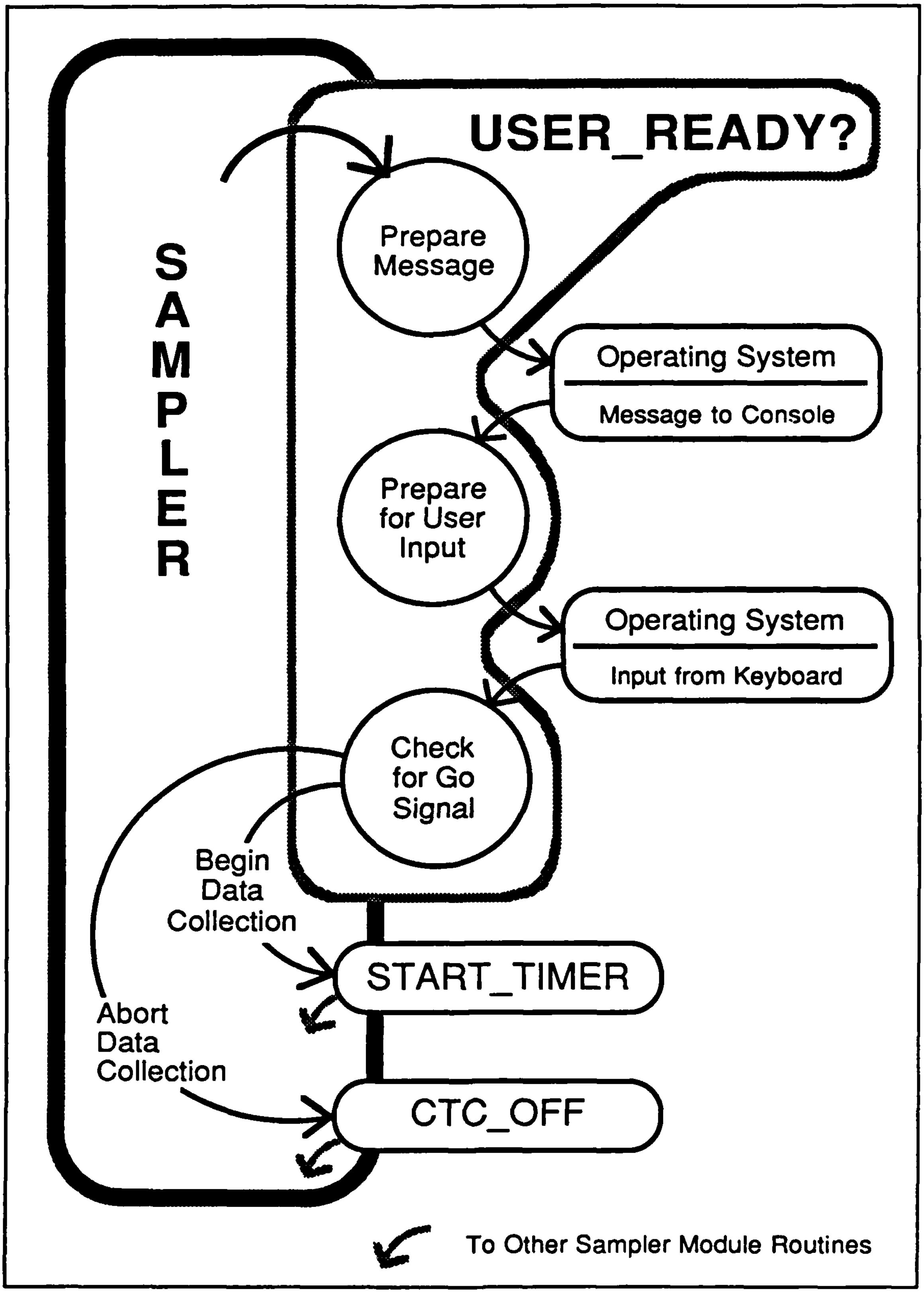


Figure 52. Program Flow within USER_READY?

6. Invocation

USER_READY? is invoked from SAMPER simply by its name.

7. Variables and Constants

a. Variables

USER_READY? uses three variables. USER_READY? loads the output parameter ERROR_CODE with either FALSE or ABORT to indicate to the calling PLZ routine whether and error abort occured or not. The second variable used is the character returned from the operating system call to the system keyboard. This variable is located in the buffer location RTN_MESS. The last variable used is not a true variable, rather it is the state of the Z-8Ø zero flag. The state of this flag is used to indicate to SAMPLER whether Sampler Module should continue execution or be terminated.

b. Constants

USER_READY? uses a host of module constants. Their names, values, and definitions follow. Of particular interest are the definitions of the Operating System Call Vector constants.

Constant Name	Value	Definition
FALSE	ØØ hex	All is OK Error Code.
ERROR_CODE	1Ø hex	IX offset for output parameter ERROR_CODE.
A_VECTOR	undefined	Beginning Address of the Buffer for the Operating System Call Vector, defined during Module linking.
A_LOGICAL_UNIT	A_VECTOR + ØØ hex	Call Vector Position for Logical Unit Desired.
A_REQUEST_CODE	A_VECTOR + Ø1 hex	Call Vector Position for the System Request Code. See WRITELN and READLN below.
A_DATA_TRANS	A_VECTOR + Ø2 hex	Call Vector Position for Pointer to Data Transfer location. See MESSAGE and RTN_MESS.

Constant Name	Value	Definition
A_BYTE_COUNT	A_VECTOR + Ø4 hex	Call Vector Position for Number of Bytes to Be Transfered.
A_RETURN	A_VECTOR + Ø6 hex	Call Vector Position for the No Error Return Address.
A_ERR_RETURN	A_VECTOR + Ø8 hex	Call Vector Position for Error Return Address
A_COMP_CODE	A_VECTOR + ØA hex	Call Vector Position for Operating System Completion Code.
CONOUT	Ø2 hex	Logical Unit Number for System Console.
WRITELN	1Ø hex	Request Code for Output.
MESSAGE	undefined	Address of first character of message "Collection system ready.. Begin ?" Address defined upon Module Linking.
L_MESSAGE	21 hex	Length of MESSAGE.
SET?	undefined	Address of a Section of USER_ READY?, used for A_RETURN and A_ERR_RETURN. Defined at Time of Module Linking.
SYSTEM	14Ø3 hex	Address of Operating System Entry Point.
CONIN	Ø1 hex	Logical Unit Number for System Keyboard
READLN	ØC hex	Request Code for input.
RTN_MESS	undefined	Address of a buffer used to receive the User's response. Defined during linking.

Constant Name	Value	Definition
GO	undefined	Address of a Section of USER_READY? used as the A_RETURN and A_ERR_RETURN. Defined at Time of Module Linking.
Y_ASCII	59 hex	The ASCII character "Y".
ABORT	AB hex	Error Code for User Aborted Data Collection.

8. Other Routines Called

USER_READY? calls the operating system to output a message and to receive user go ahead for data collection. The call to the operating system is accomplished by loading a transfer buffer know as an Operating System Call Vector with the *information required* by the *operating system*, loading the address of the buffer into the IY register, and then calling the operating system. The call vector's content is shown above in the A_VECTOR definitions in the list of constants used by USER_READY?.

9. Output of Routine

The output of USER_READY? is the status of the Z-8Ø CPU's zero flag. If the Z flag is set (a one), then the user responded with a "Y" and data collection should proceed. If the Z flag is not set (a zero), then data collection should be aborted.

10. Routine Testing

USER_READY? was tested along with the other routines of Sampler Module.

11. Reference to Listing

The listing of USER_READY? is on pages 345-346 in Appendix D.

1. Routine Name: **START_TIMER**

2. Subordinate Routine of Sampler Module

3. Written in Z-8Ø assembly language; 3 lines (6 bytes) of code.

4. Synopsis of Routine

This sole purpose of this very short routine is to supply the final third of the CTC programing begun by CTC_PROGRAM. The effect of this is to turn on the CTC timer and interrupts. START_TIMER obtains the command it writes to CTC1, channel zero, from the system stack. The command is the input parameter TIME_CNST.

5. Routine Relationships Diagram

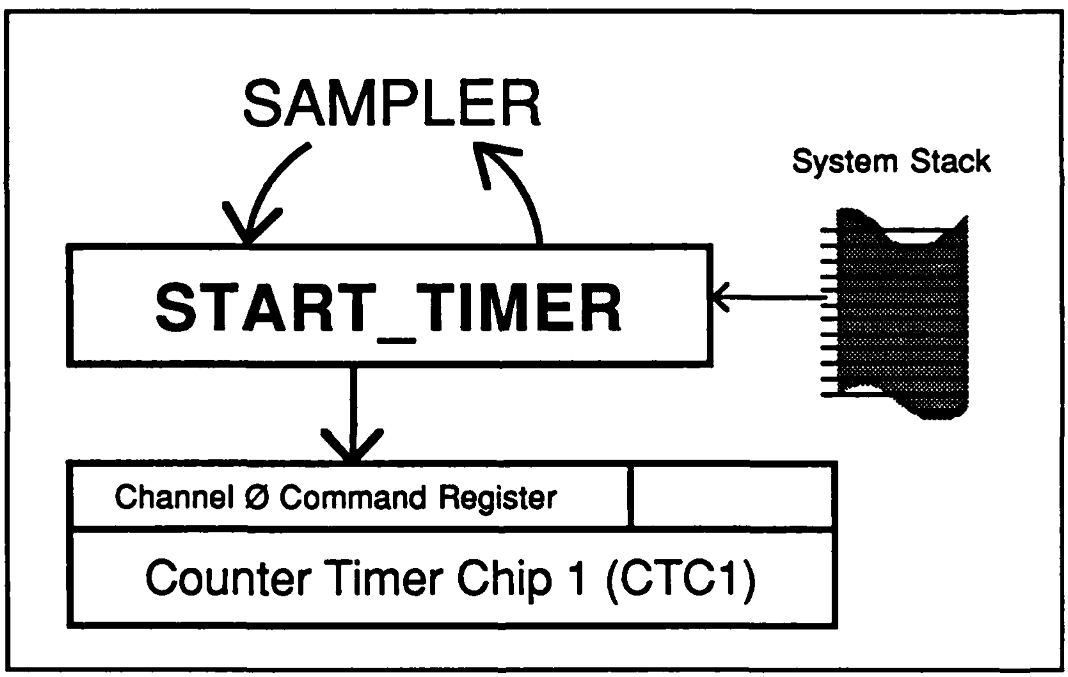


Figure 53. Relationship of START_TIMER to SAMPLER, CTC1, and the System Stack

6. Invocation

START_TIMER is invoked by name only through the Z-8Ø instruction CALL. There are no parameter passing lists in assembly language subroutine calls.

7. Variables and Constants

START_TIMER uses one variable, the input parameter TIME_CNST. This variable is obtained from the system stack via the module constant TIME_CNST (value ØA hex) which is the IX register offset to the input parameter's location on the stack. START_TIMER also uses the module constant CTC1_CMD (value 84 hex) which is the address of the CTC1, channel Ø command register.

8. Other Routines Called

START_TIMER calls no other routines.

9. Output of Routine

The impact of START_TIMER is significant. By writing the time constant to the CTC, the CTC programming is complete and it begins its timing and interrupting.

10. Routine Testing

No individual testing was performed on START_TIMER.

11. Reference to Listing

The program listing of START_TIMER can be found on page 347 in Appendix D.

1. Routine Name: **COLLECTER**

2. Subrodinate Routine of Sampler Module

3. Written in Z-8Ø assembly language; 15 lines (3Ø bytes) of code.

4. Synopsis of Routine

COLLECTER is the heart of Sampler Module. COLLECTER reads in the data from the AIO board and stores it a memory buffer. COLLECTER continues to read in data until the specified number of samples have been collected.

The executation states of COLLECTER were shown in Figure 43 in the introduction to the Sampler Module. COLLECTER primarily sits in a loop, checking the AIO board status register until the least signinficant bit becomes a one signaling that data is ready. The lower eight bits of data is then read in and stored in a tempory buffer whose address is stored in the HL register. The lower data is then transfered into the data buffer. The rather complex Z-8Ø instruction LDI handles the transfer of the data (HL & DE registers), the decrementing of the sample count (BC register) and incrementing the pointer to the next buffer location (DE register). The upper four bits are then read in (in an eight bit word), stored in a tempory buffer, and then stored in the data buffer, again with the LDI instruction. If the down counter (BC register) has not reached zero, COLLECTER returns to its AIO status register checking loop. If all the samples have been collected, COLLECTER ends, returning program execution to SAMPLER.

5. Routine Relationship Diagram

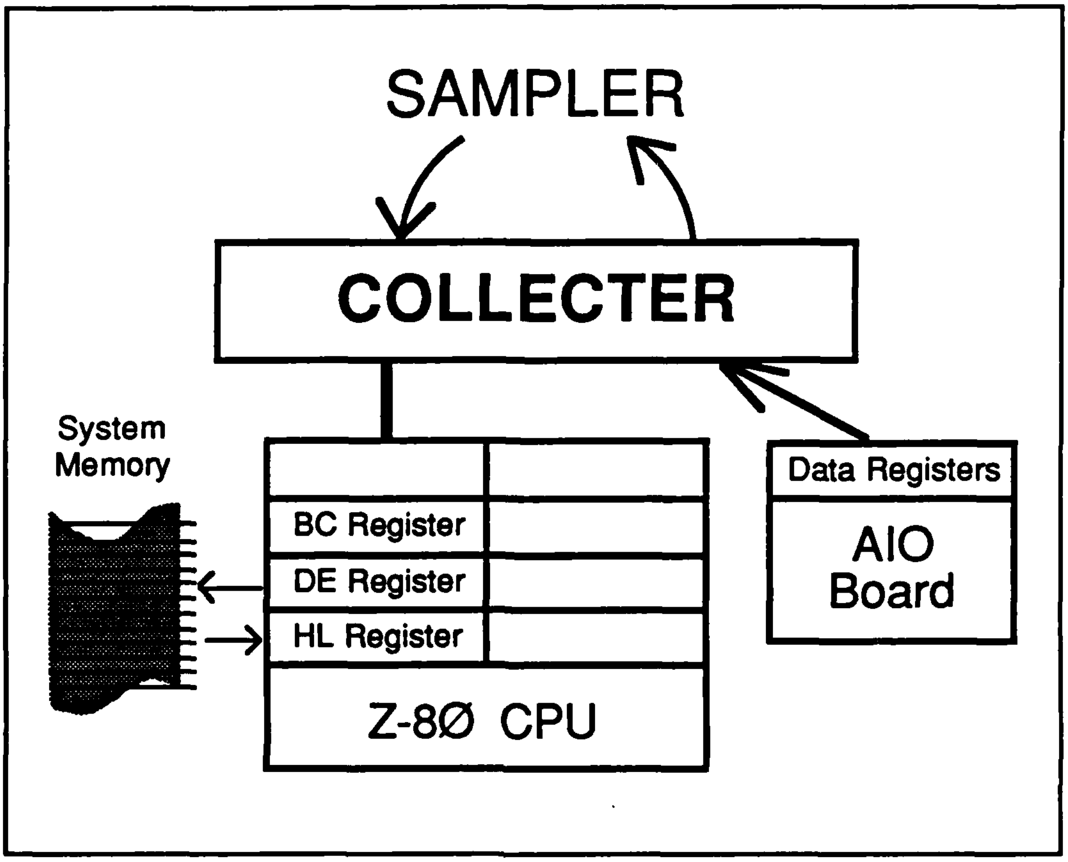


Figure 54. Relationship of COLLECTER to SAMPLER, System Memory, the Z-8Ø Primary Registers, and the AIO Board.

6. Invocation

COLLECTER is invoked by name only. There are no parameter lists.

7. Variables and Constants

a. Variables

While COLLECTER uses no named variables, the primary registers of the Z-8Ø CPU and some memory buffers are used to hold the values necessary for COLLECTER to execute. The interrupt service routine, operating concurrently with COLLECTER, uses the alternate registers of the CPU to hold the values it needs. The registers and memory buffers used by COLLECTER are

Register	Register Function - Quantity Stored
A	Receives the data from the AIO board via the IN,A instruction. The data is then placed in the temporary buffers.
BC	Holds the down counter for the number of samples. BC is loaded by INIT_COUNTER. BC is decremented by the two LDI instructions used in COLLECTER. An INC BC is included in COLLECTER to keep the BC value the number of samples, not the number of data bytes written to the memory buffer.
DE	Holds the address of the next memory buffer location. INIT_COLLECTER loads DE with the beginning address of the buffer. DE is incremented by LDI. So that DE holds the address of the lower half of the last sample stored, DE is decremented by COLLECTER upon its termination.
HL	Holds the address of the tempory buffers in which data bytes are placed. HL is loaded with the address of lower temporary buffer (DataLower) in COLLECTER's AIO status loop. The first LDI increments HL so it points to the upper temporary buffer (DataUpper).
L_BUFFER	A memory location used as a temporary buffer for the lower eight bits of data read in from the AIO board. HL holds the address of DataLower.
H_BUFFER	A memory location one above DataLower which is used as a temporary buffer for the upper data byte read in from the AIO board. After the first LDI, HL holds the address of DataUpper.

b. Constants

COLLECTER uses five module constants to refer AIO registers and buffer registers. These constants, their values, and their definitions are listed below

Constant Name	Value	Definition
L_BUFFER	undefined	The address of a memory location used for temporary storage of the lower AIO data. The value of L_BUFFER is defined when Sampler Module is linked.
H_BUFFER	undefined	The address of a memory location used for temporary stroage of the upper AIO data. This location is one above L_BUFFER.
ATODSTATUS	29 hex	The address of the AIO board status register.
DATALOWER	2Ø hex	Address of the AIO board lower data register.
DATAUPPER	21 hex	Address of the AIO board upper data register.

8. Other Routines Called

COLLECTER calls no other routines.

9. Output of Routine

COLLECTER reads in a user selected number of sixteen bit values from the AIO board and stores them in memory.

10. Routine Testing

COLLECTER was tested in concert with the rest of Sampler Module.

11. Reference to Listing

The code listing for COLLECTER is on page 348 in Appendix D.

1. Routine Name: **CTC_OFF**

2. Subordinate Routine of Sampler Module

3. Written in Z-8Ø assembly language; 5 lines (7 bytes) of code.

4. Synopsis of Routine

The sole purpose of this little routine is to turn the CTC timing and interrupting off. Is is accomplished by writing the off command to the command register of CTC number one (CTC1). Prior to writing to the CTC, Z-8Ø interrupts are disabled to prevent inadvertant interrrupts. Z-8Ø interrupts are enabled by CTC_OFF prior to its return to the calling routine, SAMPLER.

5. Routine Relationship Diagram

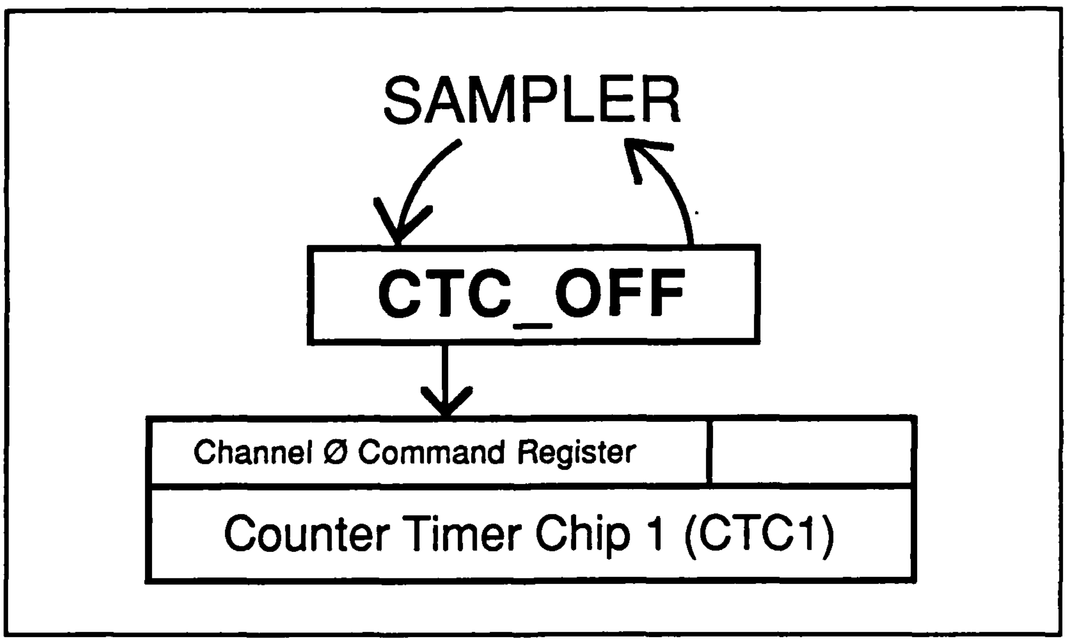


Figure 55. Relationship of CTC_OFF to SAMPLER and the CTC.

6. Invocation

CTC_OFF is invoked by name. It has neither input nor output parameters.

7. Variables and Constants

CTC_OFF uses no variables. It does use the following two global constants.

Constant Name	Value	Definition
CMD_CTC_OFF	78 hex	Command to hault and disable interrupts
CTC1_CMD	84 hex	Command port address for CTC1, channel Ø

8. Other Routines Called

CTC_OFF calls no other routines.

9. Output of Routine

CTC_OFF has no output. Its impact on system configuration is to turn off the CTC1, channel Ø timer and inhibit CTC1 from issuing interrupts.

10. Routine Testing

A variant of CTC_OFF was successfully used in another program yielding some faith that CTC_OFF would function properly. For this effort CTC_OFF was tested in conjuction with the rest of Sampler Module as described in the module discussion.

11. Reference to Listing

The program listing of CTC_OFF is on page 349 in Appendix D.

1. Routine Names: **TO_SAMPLE** and **TC_SAMPLE**

2. Interrupt Service Routines of Sampler Module

3. Written in Z-8Ø assembly language;
   TO_SAMPLE: 4 lines (6 bytes) of code;
   TC_SAMPLE: 19 lines (25 bytes) of code.

4. Synopsis of Routine

TO_SAMPLE or TC_SAMPLE is the interrupt service routine for the Sampler Module. TO_SAMPLE, for "Timer Only", is used for timer periods between 5Ø microseconds and 1Ø milliseconds. TC_SAMPLE, for "Timer and Counter, is used for timer periods between 10 milliseconds and 29.3 minutes. Which routine is used is determined by INT_SET_UP based on the input parameters to Sampler Module. INT_SET_UP loads the starting address of the selected routine into the interrupt jump table. The two routines service the CTC timer interrups differently.

TO_SAMPLE swaps CPU AF register banks, outputs to the AIO channel select port the desired analog input channel, swaps the AF register banks back, and then returns from the interrupt. The register banks are swaped to gain

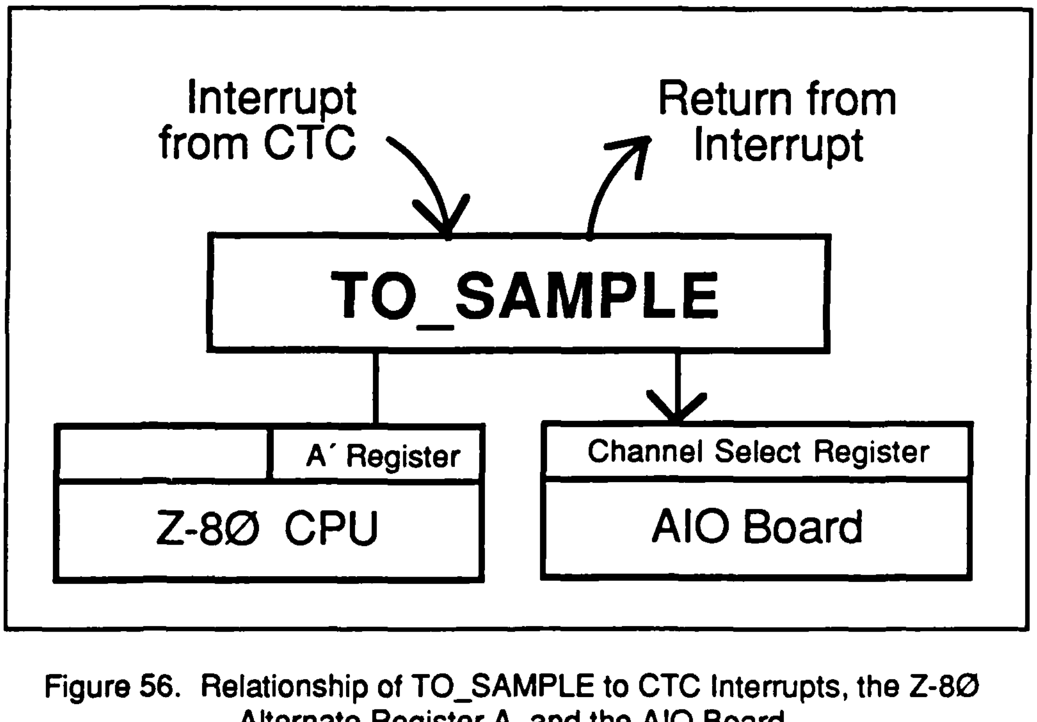


Figure 56. Relationship of TO_SAMPLE to CTC Interrupts, the Z-8Ø Alternate Register A, and the AIO Board.

access to the A register of the alternate register set which holds the desired analog channel number and to prevent interference with COLLECTER. By selecting an AIO input channel, an analog to digital conversion is initiated on that channel.

TC_SAMPLER is more complicated. To achieve the longer timing periods, TC_SAMPLER has a sixteen bit counter decremented by each interrupt. When called, TC_SAMPLE first swaps the AF, BC, DE, and HL registers to protect the contents of the primary bank of registers and to gain access to the counter values stored in the alternate bank of registers. The counter is then decrement. When the counter reaches zero, TC_SAMPLE writes the desired analog input channel number to the AIO board, initiating an analog to digital conversion, and resets the counters. Just prior to returning from interrupt, TC_SAMPLE swaps the primary register bank back.

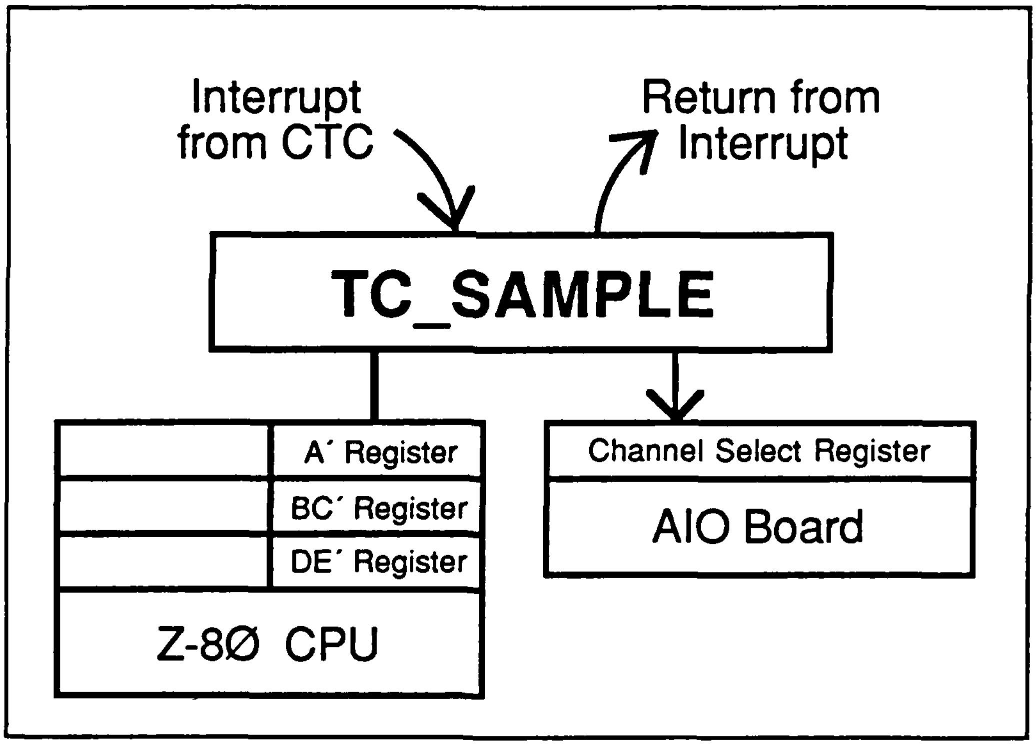


Figure 57. Relationship of TC_SAMPLE to CTC Interrupts, the Alternate Registers of the Z-8Ø CPU, and the AIO Board.

5. Invocation

Neither TO_SAMPLE nor TC_SAMPLE are "invoked." Rather, when a CTC initiated interrupt occurs, one of these routines will begin execution.

6. Variables and Constants

Neither TO_SAMPLE nor TC_SAMPLE use any named variables. Rather, these routines make use of values saved in the alternate register set of the Z-8Ø CPU. Both routines use the alternate A register (A´) to hold the number of the user specified AIO board analogue input channel. Both routines write this number to the AIO channel select register to initiate an analog to digital conversion. TC_SAMPLE also uses the alternate BC (BC´) and DE (DE´) registers. BC´ holds the current down counter value that is decremented with each call of TC_SAMPLE. DE´ holds initial value of the down counter; DE´ is used to reset BC´ when the counter reaches zero.

Both TO_SAMPLE and TC_SAMPLE use the module constant CHANNEL_SELECT value 28 hex, for the address of the AIO input channel selection register.

7. Other Routines Called

TO_SAMPLE and TC_SAMPLE call no other routines.

8. Output of Routine

In the single execution of Sampler Module, TO_SAMPLE or TC_SAMPLE can be called hundreds to thousands of times. Each time TO_SAMPLE is called, an analog to digital conversion is initiated. Each time TC_SAMPLE is called the down counter is decremented; when it reaches zero an analog to digital conversion is initiated.

9. Routine Testing

Both TO_SAMPLE and TC_SAMPLE were tested in conjunction with the rest of the Sampler Module routines. Being interrupt service routines there is no way they could be tested independently.

10. Reference to Listing

The listings of TO_SAMPLE and TC_SAMPLE are on page 350 in Appendix D.

1. Routine Name: **DEALLOCATE**

2. Subordinate Routine of Sampler Module

3. Written in Z-8Ø assembly language; 11 lines (16 bytes) of code.

4. Synopsis of Routine

DEALLOCATE is the last subordinate routine of Samper Module. DEALLOCATE handles all prepartions for the return to the calling PLZ routine. Specifically, DEALLOCATE loads the addresses of the last data values stored into the output parameter LAST_DATA's storage location in the system stack. Then DEALLOCATE pops the calling routine's IX regester value (into IX) and the return address (into HL) from the system stack. Last, DEALLOCATE pops from the system stack the storage locations for the input parameters. Having completed its actions, DEALLOCATE returns to SAMPLER.

5. Routine Relationship Diagram

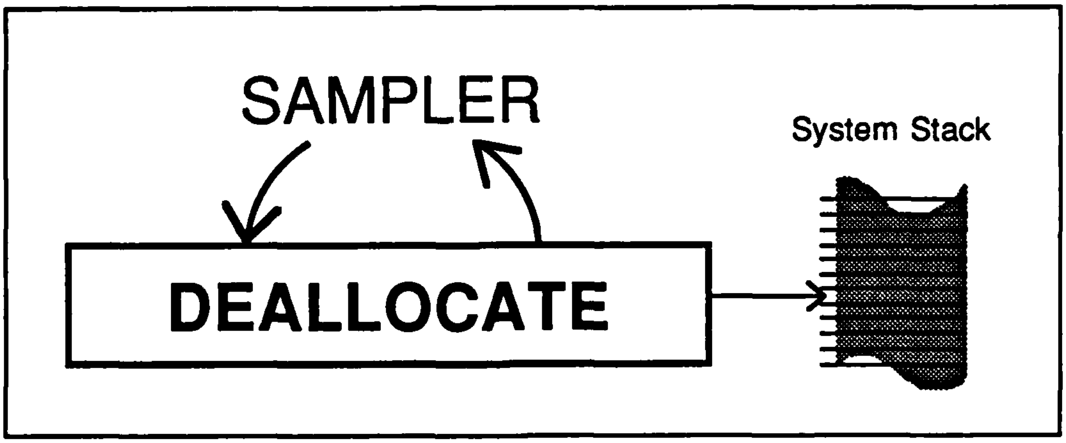


Figure 58. Relationship of DEALLOCATE to SAMPLER and the System Stack

6. Invocation

As an assembly language subroutine, DEALLOCATE is invoked by name only. There is no parameter passing.

7. Variables and Constants

DEALLOCATE uses no variables. It does use the module defined constant LAST_DATA, value 12 hex, for the IX register offset to the storage location of the output parameter LAST_DATA. DEALLOCATE does load the HL register with the return address for the calling routine.

8. Other Routines Called

DEALLOCATE calls no other routines.

9. Output of Routine

At the end of DEALLOCATE's execution, the HL register holds the calling routine's return address, the IX register holds the calling routine's original IX value, the return parameter LAST_DATA is in place in the system stack, and the input parameter storage locations in the system stack have been deallocated.

10. Routine Testing

DEALLLOCATE was tested with the rest of Sampler Module routines.

11. Reference to Listing

The program listing of DEALLOCATE is on page 351 in Appendix D.

This page is intentionally blank.

V. Buffers Module

Definition of Buffers Module

Buffers Module is unique among the modules of the data collection system in that it contains not one line of code. Rather than code, the Buffers Module holds the definition of the memory buffer that Sampler Module loads data into and that routine LOAD_DATA_FILE, of Collect_Data Module, reads data from and writes to a disk file. The global buffer established in Buffers Module is named DATA_BUFFER. It's declaira- tion statement sizes DATA_BUFFER as an array of BUFFER_SIZE words (sixteen bits). BUFFER_SIZE is a Buffers Module constant having a value of 1ØØØ decimal. Thus, DATA_BUFFER holds 2ØØØ bytes. When the whole data collection system is linked together, Buffers Module is the last module linked in.

The listing for Buffers Module is in Appendix E.

VI. Collect Data Module

Introduction to Collect_Data Module

Collect_Data Module is a set of PLZ language routines which, along with some external routines, implements a portion of a data collection system. The portion implemented is the reading in of data from the AIO board and storage of that data in a disk file. Collect_Data Module is intended to be called from a high level user interface routine.

The routines of Collect_Data Module presented here are not completely developed. They have not been assembled nor linked in with the external routines called. These routines do fully represent the design of the data collection system.

In the following sections the organization and function of the routines of Collect_Data Module will be presented. Following that will be a listing of the external routines used, a description of the invocation of Collect_Data, the variables and constants of Colect_Data Module, and the known flaws in the module. Descriptions of the fifteen routines of Collect_Data Module are then presented.

Organization of Collect_Data Module

The fifteen routines of Collect_Data Module and the thirteen external routines used by Collect_Data are organized into a hierarichal structure. There is one executive routine, SAMPLE_DATA, which calls seven subordinate routines. Five of these routines are primary subordinate routine ; they control the five major functions of Collect_Data. The routines of Collect_Data and their functions are listed below. The numbered routines are the primary subordinate routines.

Routine Name	Function
SAMPLE_DATA	Executive routine of Collect_Data Module.
GET_DATE	Via an external routine, reads the system date and loads the six characters into a string.
1. PREPARE_COLLECTOR	Finds programming commands for the CTC, the down counter value, and sizes the data buffer.

Routine Name	Function
FIND_TIME_CNST	Rounding division routine to find the CTC time constant.
FIND_CTC_COMMANDS	Based on user inputs, calculates the three values needed to set up the CTC paced interrupts.
SIZE_DATA_BUFFER	Based on user inputs, establishes the data buffer.
ERROR_IN_PREPARE	Manages error checking and error messages for PREPARE_COLLECTOR.
2. CREATE_DATA_FILE	Opens a disk file to hold the data read in by SAMPLER; loads header information into the file.
ASCII	Translates a numeric value into the string of ASCII characters that represent it.
STRING_COPY	Transcribes a string of characters into another string.
VALID_STRING	Checks the contents of a string to ensure all characters are valid for a file name.
ERROR_IN_CREATE	Error determination and error message routine for CREATE_DATA_FILE.
3. SAMPLER	Turns on the CTC interrupts, programs the AIO analog to digital converter, and reads in data from the AIO Board into the memory buffer (external routine of Sampler Module)
ERROR_IN_SAMPLER	Checks the output of SAMPLER for errors; writes error messages to the system console.
4. LOAD_DATA_FILE	Transfers the data stored by SAMPLER in the memory buffer into the disk file opened by CREATE_DATA_FILE.
5. CLOSE_DATA_FILE	Closes the disk file holding the data.

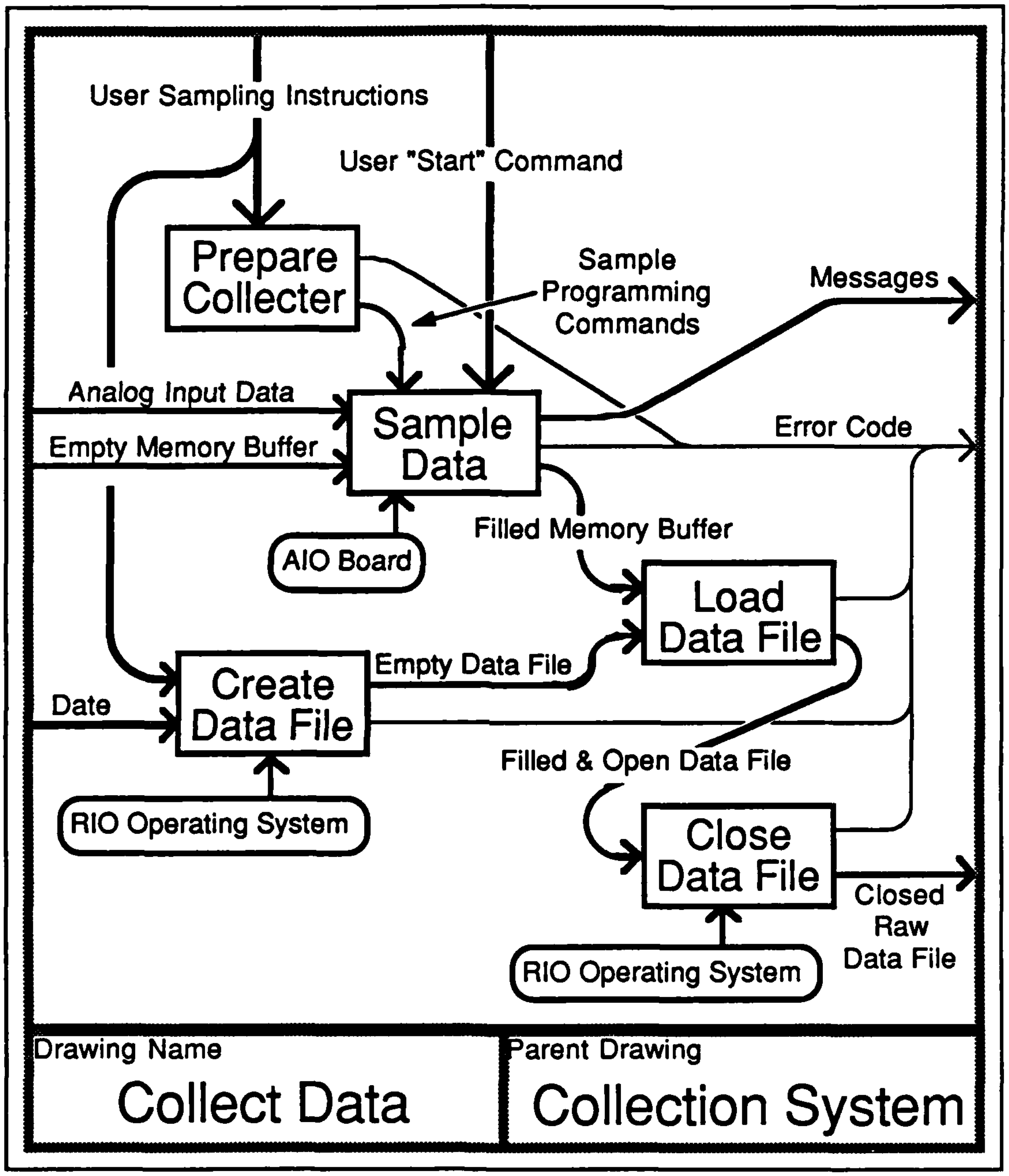


Figure 59. Data Flow Diagram for Collect_Data Module

The data flow diagram above shows the functional relationships between these five primary proccesses of Collect_Data Module. It is not coinsidence that the names of the five primary subordinate routines match these five processes. The inputs to Collect_Data Module are

the user sampling instructions,

the user "start" command,
analog input data (via the AIO board)
and the system data.

The outputs of Collect_Data (assuming all goes well) are messages written to the system console, error codes to both the system console and the calling routine, and a disk file filled with data. The sole controlling factors are the inputs from the user. The mechanisms employed to accomplish each procedures' purpose is either the RIO operating system or the AIO board.

External Routines Called By Collect_Data Module Routines

Thirteen external routines are used by Collect_Data. Their names, invocations, functions and modules are listed below.

------------------ Enhancements Module Routines --

a. WRITE_HBYTE(LOGICAL_UNIT, VALUE)

where LOGICAL_UNIT (type Byte) is the number of the device to which the hexidecimal representation of VALUE (type Byte) is to be output.

b. WRITE_HINTEGER(LOGICAL_UNIT, VALUE)

where LOGICAL_UNIT is type Byte and VALUE is type Integer. This routine is used to output the ASCII characters that form the hexidecimal representation of VALUE.

c. WRITE_DWORD(LOGICAL_UNIT, VALUE)

where LOGICAL_UNIT (type Byte) is the output device and VALUE is the number whose decimal representation in ASCII characters is to be output.

d. WRITE_RCODE(LOGICAL_UNIT, RETURN_CODE)

where both parameters are of type Byte. LOGICAL_UNIT is the number of the output device driver. RETURN_CODE is the RIO Operating System return code whose text descritption will be written to the desired device.

e. WRITELN_RCODE(LOGICAL_UNIT, RETURN_CODE)

performs the same function as WRITE_RCODE with the same parameters but adds a carriage return on the end of the text description.

f. WRITE(LOGICAL_UNIT, TEXT_POINTER)

where LOGICAL_UNIT, of type Byte, designates the device to which output is directed. TEXT_POINTER, type PByte for Pointer-To-Byte, points to the first character of the text string to be output. Characters will be output until a carriage return is encountered. The carriage return will not be output.

g. WRITELN(LOGICAL_UNIT, TEXT_POINTER)

is identical to WRITE except WRITELN does output the carriage return.

------------ PLZ.STREAM.IO Module Routines --

h. RETURN_CODE :=
OPEN(LOGICAL_UNIT, FILE_NAME_PTR, MODE)

where RETURN_CODE, LOGICAL_UNIT, and MODE are type Byte and FILE_ NAME_ PTR is type PByte. The purpose of OPEN is to open a disk file. RETURN_CODE passes back the RIO operating system completion code. LOGICAL_UNIT passes in the desired logica unit number for the file. FILE_NAME_ PTR points to the first character of a text string which holds the desired file name. MODE pases in the type of opening desired.

i. RETURN_CODE := CLOSE(LOGICAL_UNIT)

where both parameters are type Byte. CLOSE's function is to close an open disk file. RETURN _CODE passes back the operating system's code descriptor of operation performance. LOGICAL_UNIT is the logical unit number of the file to be closed.

j. RETURN_CODE := PUTSEQ(LOGICAL_UNIT, BUFFER_PTR,
NUMBER_OF_BYTES)

where RETURN_CODE and LOGICAL_UNIT are type Byte, BUFFER_PTR is type PByte, and NUMBER_OF_BYTES is type Word. PUTSEQ outputs the string of characters (or byte values) pointed to by BUFFER_PTR. If no errors occur, NUMBER_OF_BYTES bytes will be output to the designated LOGICAL_UNIT. The return parameter RETURN_CODE passes back the operating system completion code.

------------Sampler Module Routine --

```
k.    ERROR_CODE, LAST_DATA :=
                         SAMPLER(   IO_CHANNEL, CTC_MODE,
                                    TIME_CNST, COUNT,
                                    NUM_SAMPLES, FIRST_DATA  )
```

where ERROR_CODE, IO_CHANNEL, CTC_MODE, and TIME_CNST are type Byte, COUNT and NUM_SAMPLES are type Word, and LAST_DATA and FIRST_DATA are type PByte. SAMPLER is a collection of assembly language routines which activates an interrupt driven data collection effort that yields a memory buffer full of data. IO_CHANNEL is the AIO board input channel desired. CTC_MODE and TIME_CNST are the programming values for the CTC chip. COUNT is the value required for a down counter. CTC_MODE, TIME_CNST, and COUNT jointly define the sampling interval. NUM_SAMPLES is the number of 12-bit analog to digital conversion values to be read in. FIRST_DATA points to the beginning of the data buffer. Upon return, LAST_DATA points to the last data location in memory. ERROR_CODE returns a sing byte code for routine performance indications.

<u>Invocation of Collect_Data Module</u>

As indicated in the introduction, Collect_Data Module is intended to be called from a higher level user interface routine. The executive routine SAMPLE_DATA is the interface between the calling routine and Collect_Data Module. Its invocation is

```
ERROR_CODE := SAMPLE_DATA (  TESTID, USER_MESSAGE,
                             PERIOD_VALUE, PERIOD_UNITS,
                             INPUT_CHANNEL, SAMPLES      )
```

the type and purpose of these parameters is listed below.

Parameter	Type	Purpose
TESTID	ASCII_STRING	A six character string (plus a carriage return) that is a unique identifier for the data file, a test identifier.
USER_MESSAGE	ASCII_STRING	A free field string of characters (up to 32) of user message for inclusion in the data file.
PERIOD_VALUE	Integer	The number of time units (units given by PERIOD_UNITS) in the sampling period.
PERIOD_VALUE	Integer	The units of PERIOD_VALUE. Three are defined; microseconds, milliseconds, and seconds.
INPUT_CHANNEL	Byte	The AIO board input channel (Ø-15) to be used.
SAMPLES	Word	The number of data samples to be collected.
ERROR_CODE	Byte	A one byte code passed back to indicate the degree of success of Collect_Data Module.

For SAMPLE_DATA to be called and function, Collect_Data, Enhancements, Sampler, and PLZ.STREAM.IO modules must all be linked in with the calling routine.

Collect_Data Module Variables and Constants

There are no module level variables used by any of the Collect_Data routines. Other than the input / output parameters and the global buffer DATA_ BUFFER, no global variables are used by any module routines. Quite a few constants are used however. Their names, values, and definitions are listed on the following page.

Constant Name	Value	Definition
MICRO_SECONDS	-6 dec	A possible value for the input parameter PERIOD_UNITS.
MILLI_SECONDS	-3 dec	A value for the input parameter PERIOD_UNITS.
SECONDS	Ø dec	Third possible value for PERIOD_UNITS
SLOW_MODE	87 hex	A programming word for the CTC indicating an interrupting timer with a prescale factor of 256. It is one of the possible values passed to SAMPLER via its input parameter CTC_MODE.
FAST_MODE	A7 hex	A programming word for the CTC indicating an interrupting timer with a prescale factor of 16. It is one of the possible values passed to SAMPLER via its input parameter CTC_MODE.
END_OF_STRING	7C hex	The ASCII character " | " which is used to indicate end of string.
END_OF_FILE	FF hex	MCB standard end of file designator.
MINIMUM_TIME	5Ø dec	The minimum number of microseconds permitted for the sampling period.
CONSOLE_OUT	2 hex	The logical unit number for the system screen.
DATA_FILE	7 hex	The logical unit number chosen for the disk file.
BUFFER_SIZE	1ØØØ hex	An arbitrarily selected maximum for the data buffer.
MAX_BUFFER_ADDRESS	9AØØ hex	The upper memory address allowable for the data buffer. The value is based on where the operating system and the data collection routines are loaded.

Constant Name	Value	Definition
-------- Error Codes of Collect_Data Module --------		
FALSE	ØØ hex	No error.
FATAL	FE hex	Things have gone very wrong. Fatal error.
ABORT	AB hex	The user has signaled to hault data collection.
TOO_MANY_SAMPLES	EØ hex	The user specified more samples than there is buffer space for.
BAD_CHARACTER	BC hex	A character in a file name string is invalid.
PERIOD_RANGE_ERROR	E1 hex	The user specified the sampling interval improperly or selected an invalid range.
REDO	22 hex	The user input was not correct.
STORAGE_ERROR	23 hex	Something went wrong during the transfer of data from the memory buffer to the disk file.

Constant Name	Value	Definition
--------- RIO Operating System Return Codes Used by Collect_Data ---------		
OPERATION_COMPLETE	8Ø hex	The requested action was successfully executed.
DUPLICATE_FILE	DØ hex	The file name passed during an open new file operation already exists.
INSUFFICIENT_MEMORY	4A hex	A memory manager return if a memory allocation request cannot be satisfied.

Constant Name	Value	Definition
DEVICE_NOT_READY	C2 hex	Code for a device, such as a disk drive, being unable to respond.
FILE_NOT_FOUND	C7 hex	Return for an OPEN request, other than create, when the desired file isn't on the disk directory.

Note: no constants are defined at the routine level.

Flaws in Collect_Data Module

Aside from the fact that this module was never assembled, there are a number of flaws present in Collect_Data Module. Most of these flaws are presented in the discussions of the individual routines. Two errors are present in the module overhead however. First, in the introductionary comments, the third routine listed should be SAMPLER not SAMPLE_DATA. SAMPLE_DATA is the executive routine for Collect_ Data Module. The second error is more serious. In the externals definition section, the order of parameters for SAMPLER is in error. The SAMPLER definition should appear as

```
SAMPLER   PROCEDURE(  IO_CHANNEL  CTC_MODE  TIME_CNST   BYTE,
                      COUNT  NUM_SAMPLES   WORD,
                      FIRST_DATA  PBYTE              )

          RETURNS (   ERROR_CODE BYTE, LAST_DATA   PBYTE  )
```

In attition to these two specific flaws, the comments of the Collect_Data Module routines just is not sufficient. This is particularly true of the later routines. Last, some of the constants defined for Collect_Data Module and one external routine (SEEK) are not used by the module.

Content of Detailed Routine Descriptions

In the following pages are detailed descriptions of the fifteen routines of the Collect_Data Module. In each description, the following information will be presented.

1. Routine Name
2. Name of Module and Role of Routine
3. Language and Length of Routine

4. Synopsis of Routine
5. Diagram of Routine Relationships
6. Invocation of Routine
7. Variables and Constants Used
8. Other Routines Called
9. Output of Routine
10. Flaws in the Routine
11. Reference to the Routine Program Listing

The program listings of Collect_Data Module are in Appendix F.

1. Routine Name: **STRING_COPY**

2. Internal routine of Collect_Data Module.

3. Written in PLZ, seven lines of code.

4. Synopsis of Routine

Procedure STRING_COPY transcribes a string of ASCII characters from one memory location to another. Since PLZ cannot directly refer to absolute memory addres-ses, pointers to the source and destination strings are used. The beginning of the source string is pointed to by the input parameter SOURCE; the beginning of the destination string location is pointed to by the input para meter DESTINATION. The transcription begins by copying the character at location S_INDEX of SOURCE to location D_INDEX of DESTINATION where S_INDEX and D_INDEX are offsets from the beginnning of the strings. Both S_INDEX and D_INDEX are input parameters to Procedure String_Copy. Transcription continues character by character until the ASCII character "|" (7C hex) is copied from SOURCE to DESTINATION. The "|" is thus used as an end of string de-limeter and is the module constant END_OF_STRING.

5. Routine Relationship Diagram

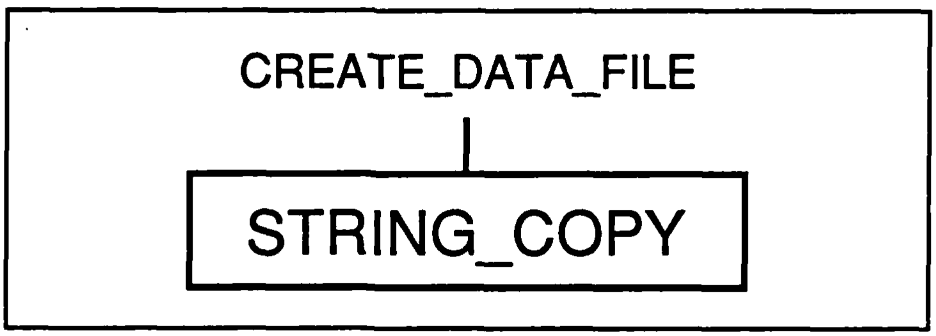


Figure 60. Relationship Between STRING_COPY and CREATE_DAT_FILE

6. Invocation

STRING_COPY is invoked from CREATE_DATA_FILE with

STRING_COPY(SOURCE, S_INDEX, DESTINATION, D_INDEX) where SOURCE and DESTINATION are of type ASCII_PTR (a pointer to an ASCII string) and S_INDEX and D_INDEX are of type byte. S_INDEX indicates which character in the SOURCE string is the first to be transcribed to the D_INDEX position in the DESTINATION string.

7. Variables and Constants

a. Global

No global variables or constants are used by STRING_COPY.

b. Internal to the Module

Beyond the input and output parameters, STRING_COPY uses no module level variables. The module constant END OF STRING, value 7C hex the ASCII character "|", is used to indicate end of string.

c. Internal to the Routine

STRING_COPY uses no routine level variables or constants.

8. Other Routines Called

STRING_COPY calls no other routines.

9. Output of Routine

Upon the completion of STRING_COPY the contents of the source string has been copied to the destination string.

10. Routine Flaws

STRING_COPY is completely acceptable in its current form.

11. Reference to Listing

STRING_COPY's program listing is on page 374 in Appendix F.

1. Routine Name: **ASCII**

2. Internal routine of Collect_Data Module

3. Written in PLZ; 28 lines of code.

4. Synopsis of Routine

ASCII takes value and translates it into a string of ASCII characters that represents the value. Also input to ASCII is the base of desired representation. Thus ASCII can be used to translate the input value into binary, decimal, or hexidecimal strings. This ASCII routine of Collect_Data Module is a combination of the ASCII and PLACE_LOOP routines of Enhancements Module. The difference between this ASCII and the combination of the Enhancements Module routines is that ASCII puts the individual characters into an string where the Enhancements Module combination writes each individual character to a desired logical unit.

ASCII accomplishes its task with a loop and a large Case statement. The loop steps through each place of the output representation, beginning with the most siglnificant place. For example, if the number 274 was to be represented in decimal, the first place to be checked would be the 1ØØ's. The contribution of each place to the total value is determined and translated into a character by a sixteen possibility ("Ø" to "9" and "A" to "F") Case statement and the character is placed in the output string. If the contribution is outside the define characters, a "?" is placed in the output character string. The loop then drops to the next significant character (or place) and determines the next contribution. The looping continues until the 1's place has been determined. The return ends by placing a carriage return on the end of the string of characters.

5. Routine Relationship Diagram

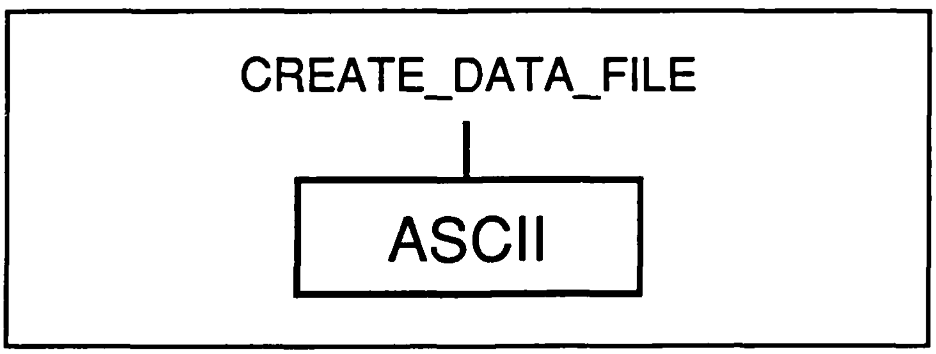


Figure 61. Relationship of ASCII and CREATE_DATA_FILE

6. Invocation

ASCII is called only by CREATE_DATA_FILE and is invoked with

CHANNEL := ASCII(**WORD**(INPUT_CHANNEL), 1Ø, 1Ø, CHANNEL)

which corresponds to the ASCII parameter definitions

TEXT_STRING := ASCII(NUMBER, INDEX, DIVISOR, INPOINTER)

TEXT_STRING and INPOINTER are of type ASCII_PTR (or pointer to ASCII string) and NUMBER, INDEX, and DIVISOR are of type Word. INPOINTER^[Ø] passes in the starting location of the output string. Strictly speaking, the return parameter TEXT_STRING isn't necessary. It was included to make clear the output of the routine. NUMBER is the value to be translated into its character string representation. DIVISOR is the base of the representation, and INDEX is DIVISOR raised to the highest anticipated factor.

7. Variables and Constants

Two locally defined variables, VALUE and POINT, are used by ASCII. VALUE, of type Word, holds the value contributed to NUMBER by each place of the character string representation. VALUE is obtained by integer division of NUMBER by INDEX. POINT, of type Byte, is a place keeper for the current location in the output TEXT_STRING. POINT is incrimented for each character or place.

ASCII uses one constant, CARRIAGE_RETURN, to represent the ASCII carriage return (value ØD hex).

8. Other Routines Called

ASCII calls no other routines.

9. Output of Routine

At the end of ASCII, TEXT_STRING is filled with the characters that represent the value of NUMBER in base DIVISOR.

10. Routine Flaws

ASCII is acceptable in its current form.

11. Reference to Listing

The program listing of ASCII is on page 374 - 375 in Appendix F.

1. Routine Name: **GET_DATE**

2. Internal routine of Collect_Data Module

3. Written in PLZ; 1Ø lines of code.

4. <u>Synopsis of Routine</u>

Procedure GET_DATE interfaces Collect_Data Module with the Utility Module, asembly language routine DATE. DATE obtains the current system date from its storage location in memory and passes back six Byte valued, the six characters representing the day, month, and year. GET_DATE takes these six characters and places them into a single ASCII string. The releationships of these routines is shown in the figure below.

5. <u>Routine Relationship Diagram</u>

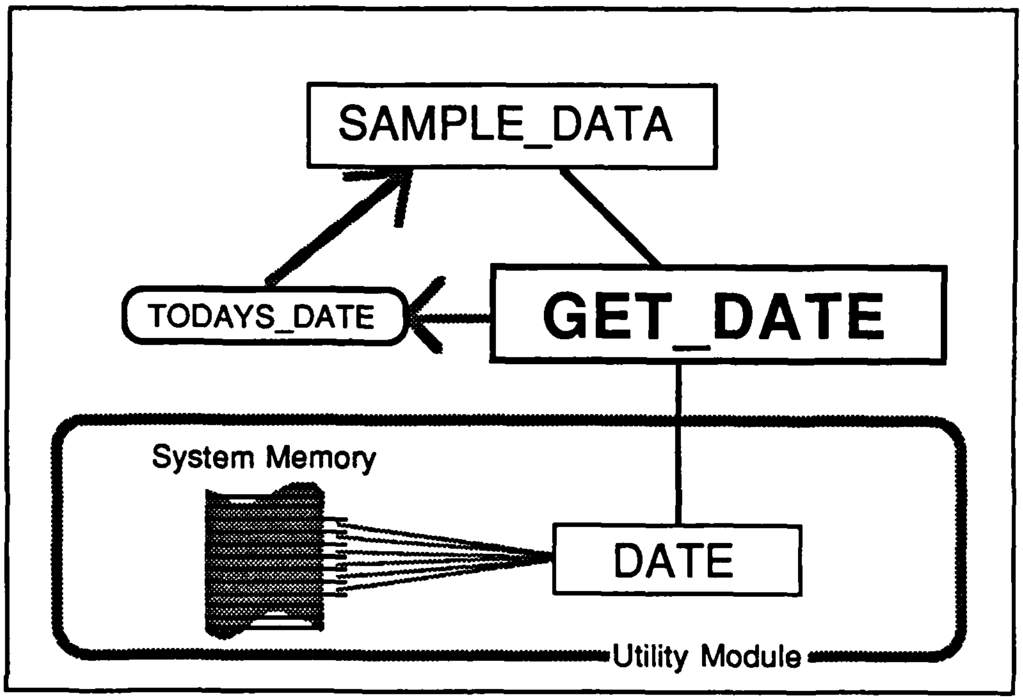


Figure 62. Relationship of GET_DATE to SAMPLE_DATA and DATE.

6. Invocation

GET_DATE is called only by SAMPLE_DATA and is invoked with

TODAYS_DATE := GET_DATE(IN_POINTER)

where both TODAYS_DATE and IN_POINTER are both of type ASCII_PTR for pointer to ASCII string. The output parameter TODAYS_DATE isn't really necessary as IN_ POINTER supplies all the information necessary for GET_DATE to load the character string. TODAYS_DATE was included to make clear the output of the routine.

7. Variables and Constants

GET_DATE uses six local Byte valued variables. These six variables, YEAR1, YEAR2, MONTH1, MONTH2, DAY1, and DAY2 are used for the return parameters in the call to the external routine DATE. GET_DATE uses one module level constant, CARRIAGE_RETURN, of value ØD hex.

8. Other Routines Called

GET_DATE calls DATE, and external routine of the Utility Module, to get the six characters of the system date. DATE is invoked with

YEAR1, YEAR2, MONTH1, MONTH2, DAY1, DAY2 := DATE

where each of the six output parameters are of typeByte and hold an ASCII character.

9. Output of Routine

GET_DATE results in the text string TODAYS_DATE begin filled with the six characters of the system date, ending with a seventh character, a carriage return.

10. Routine Flaws

GET_DATE 's current implementation is acceptable.

11. Reference to Listing

The listing of GET_DATE is on page 375 in Appendix F. The listing of DATE is in the Enhancements Module section.

1. Routine Name: **FIND_TIME_CNST**

2. Internal routine of Collect_Data Module.

3. Written in PLZ; 5 lines of code.

4. Synopsis of Routine

This little routine is used to more accurately find the CTC programming time constant. Normally, division in PLZ produces a truncated result rather than the more accurate rounded result (Ref 6: 43). FIND_TIME_CNST, via an intermediate term and modulo division, determines whether the best time constant is the truncated division (equivallent to rounding down) or should be incremented by one (equivallent to rounding up). The rounded TIME_CNST is then returned to the calling routine FIND_CTC_COMMANDS.

5. Routine Relationship Diagram

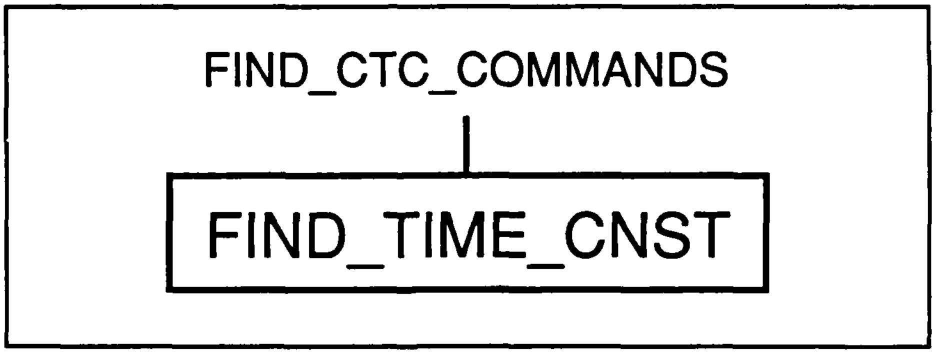


Figure 63. Relationship of FIND_TIME_CNST to FIND_CTC_COMMANDS

6. Invocation

This routine is invoked from FIND_CTC_CMDS with

TIME_CNST := FIND_TIME_CNST(TIME, MULTIPLER, DIVISOR)

where the input parameters are all of type Word and the output parameter is of type Byte. TIME corresponds to Period_Desired, MULTIPLER corresponds to Clock_Rate, and DIVISOR corresponds to Prescale_Counter. FIND_CTC_COMMANDS is carefull not to pass to FIND_TIME_CNST input parameters which would cause overflow.

7. Variables and Constants

FIND_TIME_CNST uses one internal variable, INTERMEDIATE, of type Word. INTERMEDIATE holds the product of the Period_Desired and the Clock_Rate.

8. Other Routines Called

FIND_TIME_CNST calls no other routines.

9. Output of Routine

FIND_TIME_CNST passes back to FIND_CTC_COMMANDS the time constant required to achieve the desired timing period given the CTC prescale counter value.

10. Routine Flaws

FIND_TIME_CNST is acceptable though it perhaps should be named ROUNDING_DIVISION to better reflect its basic function rather than its employment.

11. Reference to Listing

The listing of FIND_TIME_CNST's code is on page 376 in Appendix F.

1. Routine Name: **FIND_CTC_COMMANDS**

2. Internal routine of Collect_Data Module.

3. Written in PLZ; 35 lines of code.

4. Synopsis of Routine

This routine determines the values of the three parameters required to establish the desired sampling interval. Two parameters are needed to program the Counter Timer Chip (CTC) which issues periodic interrupts, the prescale counter and the time constant (Ref 7: Sec 3.7). One parameter is required for the additional sixteen bit down counter used for longer sampling intervals. FIND_CTC_COMMANDS uses FIND_TIME_CNST to determine the CTC time constant. The overall formula for the sampling interval is

Sampling Period =
Clock_Period X Prescale_Counter X Time_Constant X Counter

where

Clock_Period	=	Ø.4072 microseconds,
Prescale_Counter	=	16 or 256,
Time_Constant	=	Ø to 255, and
Counter	=	1 to 65535.

Since the user selectes the sampling period and the clock period is fixed, the three variable parameters available to FIND_CTC_COMMANDS are the prescale counter, time constant, and counter value. As discussed in the introduction to the Sampler Module, the timing periods have been divided into four ranges. Figure 64 below (a duplicate of the Sampler Module figure) shows the ranges. Within these ranges the Prescale_Factor is fixed; within the longest two ranges the Time_Constant is also fixed. The sampling period ranges and the values of the variable parameters are

Sampling Period Range	Prescale Value	Time Constant	CTC Period	Counter
minimum to 1.Ø msec	16	variable	variable	not used
1.Ø msec to 1Ø.Ø msec	256	variable	variable	not used
1Ø.Ø msec to 1.Ø sec	16	154	1 msec	variable
1.Ø sec to maximum period	256	24Ø	25 msec	variable

For the first two ranges, only the CTC Time_Constant needs to be determined as the counter isn't used. The time constant is a counter used by the

CTC. In advance of the time constant counter is a prescale counter of either 16 or 256 which correspond to the "fast mode" and "slow mode." Given the MCB clock rate of 2.547 Mhz the timing constant is found with (Ref 7: Sec 3.7)

Time_Constant = (Period_Desired X Clock_Rate) / Prescale_Counter

The time constants are found by calling FIND_TIME_CNST, a routine which performs rounding division rather than the standard PLZ trucating division. Depending upon the time period desired, one of four calls to FIND_TIME_CNST is used. These calls are discussed later.

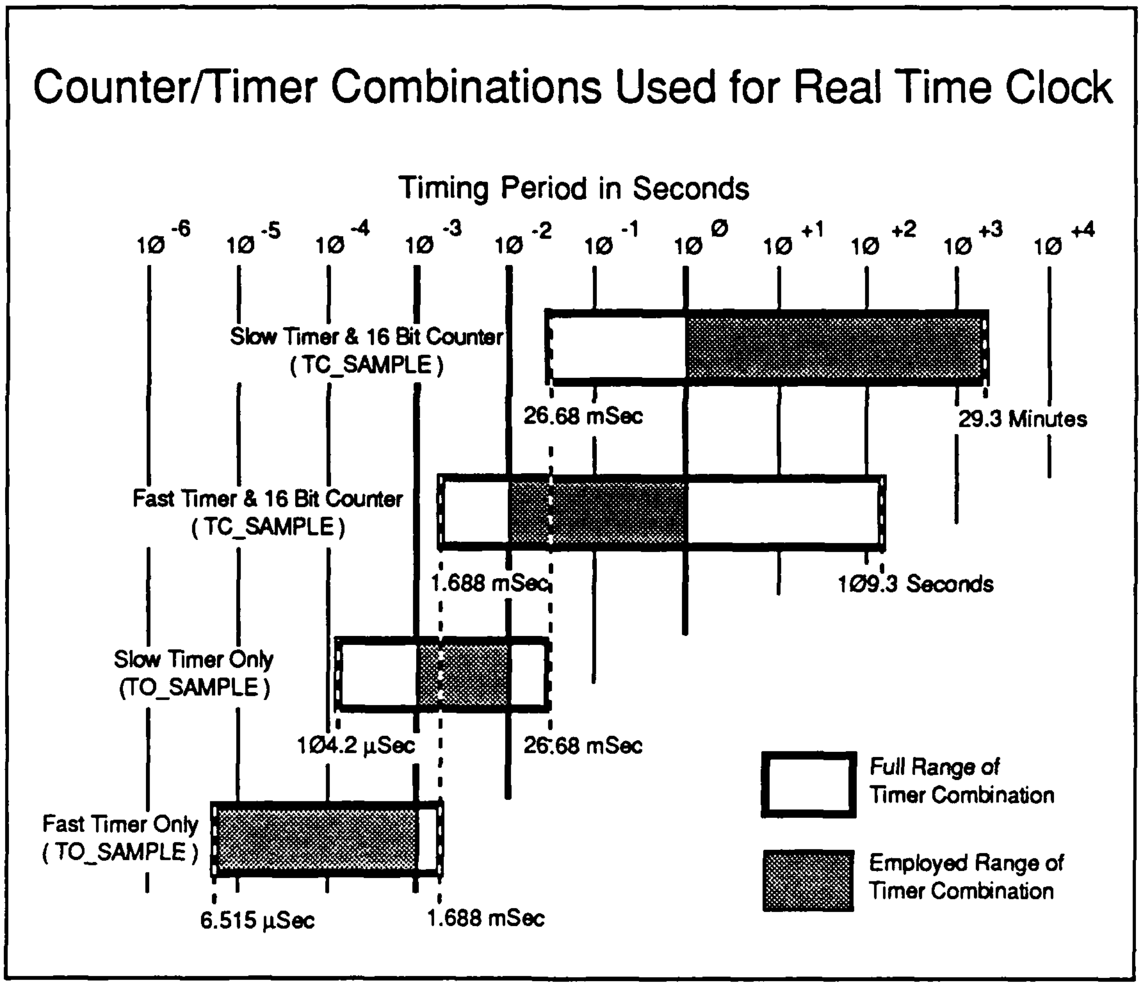


Figure 64. Counter/Timer Combinations Used for Real Time Clock

For the longer timing periods, the CTC timing is fixed and only the counter value is used to set the timing period. The formula used is

Counter := Period_Desired / CTC_Period

where CTC_Period is either 1 msec or 25 msec. In the code implementation, multiplication by the inverse of the CTC period, with adjustments for period units, is used.

Having determined the CTC_MODE, the CTC_TIME_CONSTANT, and the COUNT for the counter, FIND_CTC_COMMANDS ends.

5. Routine Relationship Diagram

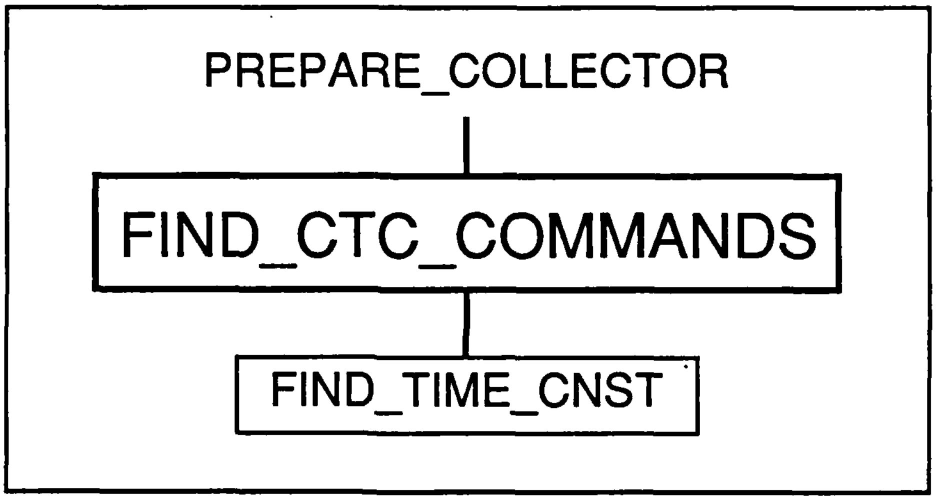


Figure 65. Relationship Between FIND_CTC_COMMANDS and PREPARE_COLLECTER and FIND_TIME_CNST

6. Invocation

FIND_CTC_COMMANDS is invoked from PREPARE_COLLECTER via

```
ERROR_CODE, CTC_MODE,
CTC_TIME_CONSTANT, COUNT :=
                    FIND_CTC_COMMANDS(  TIME,  UNITS  )
```

where TIME is the desired sampling interval measured in UNITS, both input parameters being type Integer. The output parameter ERROR_CODE, type Byte, returns an error message if an out of range sampling period was requested. CTC_

MODE, type Byte, returns the CTC command for "slow mode" (interrupting, prescale factor of 16) or "fast mode" (interrupting, prescale factor of 256). CTC_ TIME_ CONSTANT, type Byte, returns a value between 0 and 255, for the CTC counter (counter range 1 to 256). COUNT, type Word, passes back the additional counter value required for longer sampling periods. COUNT has a defined range of Ø (signaling no counter is required) to 65535.

7. Variables and Constants

FIND_CTC_COMMANDS uses no variables beyond the input and output parameters discussed above. The routine makes use of several module constants. Their names, values, and purposes are

Constant	Value	Purpose
MICRO_SECONDS	-6 dec	UNITS input to indicate units of TIME input.
MILLI_SECONDS	-3 dec	UNITS input to indicate units of TIME input.
SECONDS	Ø dec	UNITS input to indicate units of TIME input.
MINIMUM_TIME	5Ø dec	Minimum allowed microseconds for sampling.
PERIOD_RANGE_ERROR	E1 hex	Message for Out of Range Sampling Interval
FAST_MODE	87 hex	CTC command for interrupting timer with a prescale factor of 16.
SLOW_MODE	A7 hex	CTC command for interrupting timer with a prescale factor of 256.

The MINIMUM_TIME of 5Ø microseconds was selected to allow the AIO analog to digital converter to settle and allow for the interrupt service routine cycling.

8. Other Routines Called

FIND_CTC_COMMANDS calls FIND_TIME_CNST to determine the TIME_CNST. FIND_TIME_CNST is used because it performs a rounding division rather than PLZ's standard trucation division. FIND_CTC_COMMANDS contains four calls to FIND_TIME_CNST all of the form

TIME_CNST := FIND_TIME_CNST(TIME, MULTIPLER, DIVISOR)

Both MULTIPLER and DIVISOR are passed to FIND_TIME_CNST as constants using different constants for each of the four calls. The timer periods, constants, and units of TIME used are

Sampling Range			TIME Units	MULTIPLER	DIVISOR
mimimum	to	26 µsec	microseconds	2457	16ØØØ
26 µsec	to	266 µsec	microseconds	246	16ØØ
226 µsec	to	999 µsec	microseconds	25	16Ø
1 msec	to	9 msec	milliseconds	2457	256

The values passed with MULTIPLER and DIVISOR are selected to keep FIND_TIME_CNST from having a multiply overflow and maintain the maximum accuracy possible.

9. Output of Routine

At the end of FIND_CTC_COMMANDS, the three parameters necessary to program the CTC and set up the down counter have been determined. However, if an out of range sampling period was requested, and error code will be returned to the PREPARE_COLLECTER.

10. Routine Flaws

The code and organization of FIND_CTC_COMMANDS is acceptable with one exception. ERROR_CODE needs to be set to FALSE as the first executable statement. The comment lines in the code need improvement though.

11. Reference to Listing

The program listing of FIND_CTC_COMMANDS is on page 377 -378 in Appendix F.

1. Routine Name: **SIZE_DATA_BUFFER**

2. Internal routine of Collect_Data Module.

3. Written in PLZ; 1Ø lines of code.

4. Synopsis of Routine

This routine is largely a place keeper, intended to be replaced by a routine which calls the Utility Module routine ALLOCATE. SIZE_DATA_BUFFER compairs the number of samples requested by the user with the storage supplied by Buffers Module. If the number of samples is less than 1ØØØ decimal, then all is ok and the routine will proceed. Otherwise, SIZE_DATA_BUFFER will output an error message to indicate that too many samples were requested.

Ultimately, this SIZE_DATA_BUFFFER would be replaced with a routine which calls ALLOCATE, an assembly language routine which gives PLZ programs access to the RIO operating system memory manager. Though ALLOCATE, this "new" SIZE_DATA_BUFFER could make a real time request for data storage and not be limited to preformated buffers.

5. Routine Relationship Diagram

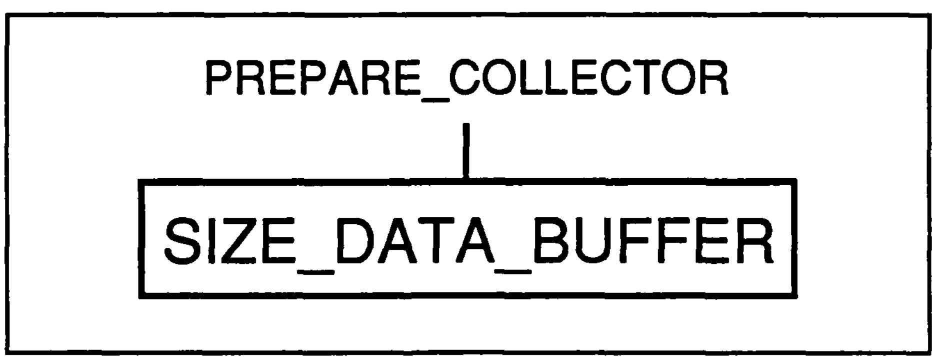


Figure 66. Relationship Between SIZE_DATA_BUFFER and PREPARE_COLLECTOR

6. Invocation

SIZE_DATA_BUFFFER is invoked from PREPARE_COLLECTER via

ERROR_CODE, SAMPLES_ALLOWED :=
SIZE_DATA_BUFFER(SAMPLES_REQUESTED)

where SAMPLES_REQUESTED and SAMPLES_ALLOWED are of type Word and ERROR_CODE is of type Byte.

7. Variables and Constants

a. Global

SIZE_DATA _BUFFER uses the globally defined DATA_BUFFER to determine how much storage area is available. The routine uses no global constants.

b. Module Level

SIZE_DATA_BUFFER uses no module level variables. SIZE_DATA_BUFFER uses three module level constants to define error codes and give the highest possible address for data in the buffer. FALSE (value ØØ hex) is the error code for no errors occured. TOO_MANY_SAMPLES (value EØ hex) is the error code output when the number of samples requested by the user exceeds AVALIABLE_WORDS. MAX_ BUFFER_ADDRESS is set to a high memory value (9AØØ hex), above the code of all the modules of the data collection system but below the system stack. This is used in conjunction with the beginning address of DATA_BUFFER to determine how much space is available for data storage (above the define range of DATA_BUFFER. This is a cludge; a call to ALLOCATE would be far superior.

c. Routine Level

SIZE_DATA_BUFFER uses a single, routine level variable AVAILABLE_WORDS to hold the number of words (one word is two bytes) available for data storage.

8. Other Routines Called

This version of SIZE_DATA_BUFFER calls no other routines. An improved SIZE_DATA_BUFFER would call ALLOCATE.

9. Output of Routine

If the number of samples requested by the user does not exceed the storage available, SIZE_DATA_BUFFER will return ERROR_CODE as FALSE and SAMPLES_ ALLOWED as the number of samples requested. However, if too many samples are requested, ERROR_CODE will be returned as TOO_ MANY_SAMPLES and SAMPLES_ALLOWED will be set to AVAILABLE_ WORDS.

10. Routine Flaws

SIZE_DATA_BUFFER is ok, but the function it performs would be far better served by calling ALLOCATE. That Utility Module routine would allow SIZE_DATA_ BUFFER to interact with the operating system memory manager.

11. Reference to Listing

The listing of SIZE_DATA_BUFFER can be found on page 379 in Appendix F.

1. Routine Name: **ERROR_IN_PREPARE**

2. Internal routine of Collect_Data Module

3. Written in PLZ; 14 lines of code.

4. Synopsis of Routine

ERROR_IN_PREPARE writes error message to the system console if an error code other than FALSE is returned by any of the routines under PREPARE_COLLECTER. Two error messages are possible. If TOO_MANY_SAMPLES is returned by SIZE_DATA_BUFFER, a message is written to the console identifying how many samples will be collected. If PERIOD_RANGE_ERROR is returned by FIND_CTC_ COMMANDS, the defined ranges will be written to the console and ERROR_MESSAGE will be reset to FATAL.

5. Routine Relationship Diagram

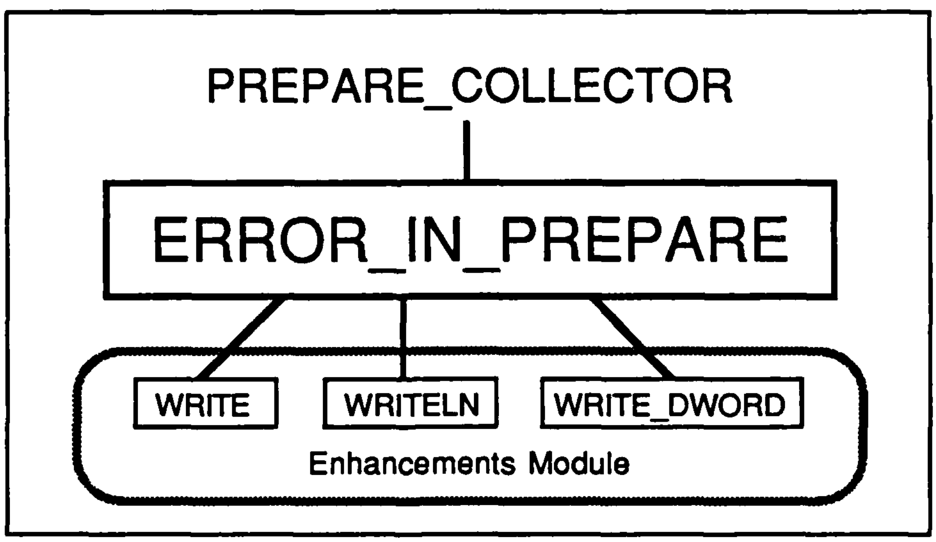


Figure 67. Relationship of ERROR_IN_PREPARE to Its Calling and Subordinate Routines.

6. Invocation

ERROR_IN_PREPARE is called from PREPARE_COLLECTER with

OUT_ERROR_CODE := ERROR_IN_PREPARE(IN_ERROR_CODE) where both the input and output parameters are of type Byte.

7. Variables and Constants

ERROR_IN_PREPARE uses no variables. It uses four constants, TOO_MANY_ SAMPLES, PERIOD_RANGE_ERROR, FALSE, and FATAL which are the possible error codes within PREPARE_COLLECTER and its subrodinate routines. The values of these module level constants are EØ hex, E1 hex, ØØ hex, and FE hex respectivally.

8. Other Routines Called

ERROR_IN_PREPARE calls three external routines, WRITE, WRITE_ DWORD, and WRITELN, all of the Enhancements Module. These three routines are used to output text strings and decimal values to the system console. The routines are invoked with

WRITE(LOGICAL_UNIT, Pointer-to-Text-String)

WRITE_DWORD(LOGICAL_UNIT, NUMBER)

WRITELN(LOGICAL_UNIT, Pointer-to-Text-String)

where LOGICAL_UNIT is of type Byte and NUMBER is of type Word. In ERROR_ IN_PREPARE LOGICAL_UNIT is all ways passed as the constant CONSOLE_ OUT. Pointer-to-Text-String could be a variable of type ASCII_PTR or could be a constant string. In ERROR_IN_PREPARE the constant string form is used. NUMBER is a sixteen bit value which WRITE_DWORD will translate into the ASCII characters of its base 1Ø representation. WRITE and WRITE_DWORD do not output carriage returns at the end of their output; WRITELN does.

9. Output of Routine

The output of ERROR_IN_PREPARE are messages to the system console which tell the user that the input parameters provided are out of range. If the error was an out of range sampling period, ERROR_IN_PREPARE returns to PREPARE_ COLLECTER the FATAL error code. Otherwise the FALSE error code is returned.

10. Routine Flaws

ERROR_IN_PREPARE's error message to the system console is wrong. It lists the mimimum time range as 7 µsec; 5Ø µsec is the correct value. Also, PREPARE_COLLECTER calls ERROR_IN_PREPARE only when errors occur. The alternate structure of having ERROR_IN_PREPARE determine whether an error has occured would be superior.

11. Reference to Listing

ERROR_IN_PREPARE's listing is on page 380 in Appendix F.

1. Routine Name: **PREPARE_COLLECTER**

2. Primary subordinate routine of Collect_Data Module

3. Written in PLZ; 13 lines of code.

<u>4. Synopsis of Routine</u>

PREPARE_COLLECTOR is the second routine called by SAMPLE_DATA, the executive routine of Collect_Data Module. PREPARE_COLLECTER takes the user supplied sampling instructions and translates them into the CTC commands and other parameters needed by Sample_Data. As shown in the figure below, PREPARE_COLLECTOR accomplishes its functions through calls to three subordinate routines, FIND_CTC_COMMANDS, SIZE_DATA_BUFFER, and ERROR_IN_PREPARE. The last routine is the error service routine for PREPARE_COLLECTER.

PREPARE_COLLECTOR is rather simple in implementation, consisting of one do loop. Within the loop, FIND_CTC_COMMANDS is called followed immediately by ERROR_IN_PREPARE to see if FIND_CTC_COMMANDS successfully executed. If an error is detected, the output error code is loaded, the do loop is exited, and PREPARE_ COLLECTOR ends. If no error occured, SIZE_DATA_BUFFER is called, again followed immediately by ERROR_IN_PREPARE. If an error is detected, th output error code is loaded, the do loop is exited, and PREPARE_COLLECTOR ends. If no error was detected, the do loop is exited and PREPARE_COLLECTOR ends. The do loop is executed only once.

5. Routine Relationship Diagram

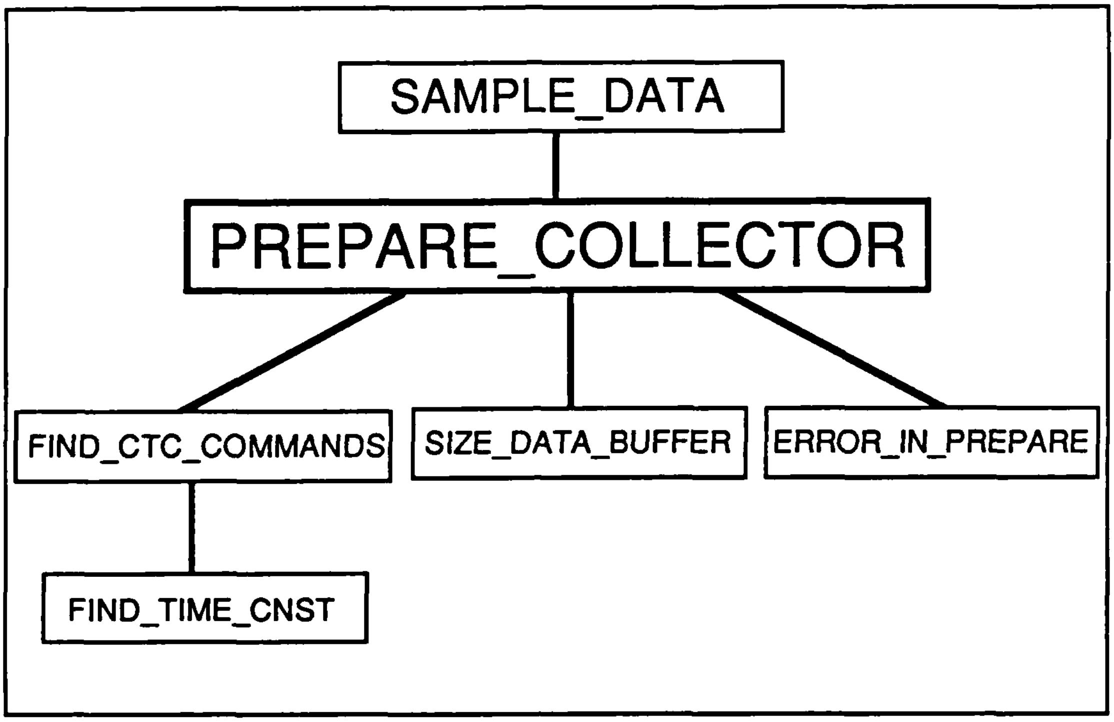


Figure 68. Relationship of PREPARE_COLLECTOR to SAMPLE_DATA and its Subordinate Routines.

6. Invocation

PREPARE_COLLECTOR is invoked from SAMPLE_DATA with

```
       ERROR_CODE,
        CTC_MODE,
     TIME_CONSTANT,
      DOWN_COUNT,
NUMBER_OF_SAMPLES =:
              PREPARE_COLLECTOR( PERIOD_VALUE,
                                 PERIOD_UNITS,
                                     SAMPLES_REQUESTED )
```

where these parameters are of the following type and purpose.

Variable Name	Type	Purpose of Parameter
---------------------- input parameters	----------------------	----------------------
PERIOD_VALUE	Integer	The desired time duration of the sampling period in PERIOD_UNITS units.
PERIOD_UNITS	Integer	The designated units of PERIOD_ VALUE. The possible values are three constants MICRO_SECONDS, MILLI_ SECONDS, and SECONDS.
SAMPLES_REQUESTED	Word	The number of analog to digital con- verisons the user wants collected and stored.
---------------------- output parameters	----------------------	----------------------
ERROR_CODE	Byte	A code to tell SAMPLE_DATA how things went within PREPARE_COL- LECTOR. Two values are possible, the constants FATAL and FALSE.
CTC_MODE	Byte	The first of two commands to the CTC to program its interrupts. CTC_MODE has two possible values SLOW_MODE and FAST_MODE.
TIME_CONSTANT	Byte	The second CTC programming com- mand. It tells the CTC how many times to count before interrupting. Values of Ø to 255 are possible with Ø meaning to count 256 times.
DOWN_COUNT	Word	The number of interrupts the down counter (used longer sampling periods) must receive before commanding the AIO board to initiate an A to D conversion.
NUMBER_OF_SAMPLES	Word	The number of samples to be collected.

7. Variables and Constants

PREPARE_COLLECTOR uses no variables beyond the input and output parameters discussed above. PREPARE_COLLECTOR uses two module level constants, FATAL and FALSE, as error codes. Values: FE hex and ØØ hex.

8. Other Routines Called

PREPARE_COLLECTER, as shown in the figure above, calls three subordinate routines FIND_CTC_COMMANDS, NUMBER _OF_SAMPLES, and ERROR_ IN_PREPARE. Their invocation statements follow.

```
ERROR_CODE, CTC_MODE, TIME_CONSTANT, DOWN_COUNT :=
        FIND_CTC_COMMANDS( PERIOD_VALUE, PERIOD_UNITS )

ERROR_CODE, NUMBER_OF_SAMPLES :=
        SIZE_DATA_BUFFER( SAMPLES_REQUESTED )

ERROR_CODE := ERROR_IN_PREPARE( ERROR_CODE )
```

Please consult the descriptions of these routines for more details.

9. Output of Routine

There are two sets of possible outputs for PREPARE_COLLECTOR. If something went seriously wrong, PREPARE_COLLECTOR will return the FATAL error code. This will cause termination of SAMPLE_DATA. If nothing went seriously wrong, PREPARE_COLLECTER will return a FALSE error code and the programming values for the CTC, down-counter, and the number of samples to be collected.

10. Routine Flaws

As it stands PREPARE_COLLECTOR is ok. It might be better to call a modified ERROR_ IN_PREPARE after each subroutine call and then check the returned error code.

11. Reference to Listing

The listing of PREPARE_COLLECTER's code is on page 381 in Appendix F.

1. Routine Name: **ERROR_IN_CREATE**

2. Subrodinate routine of Collect_Data Module.

3. Written in PLZ; 16 lines of code.

4. Synopsis of Routine

ERROR_IN_CREATE is one of the support routines for CREATE_ DATA_FILE. This routine checks the error code generated during CREATE_ DATA_FILE, outputs messages to the system console based on the error codes, and sets the final error code. As ERROR_IN_CREATE is called only if a FATAL error occurs, invocation of this routine signals termination of Collect_Data Module execution.

5. Routine Relationship Diagram

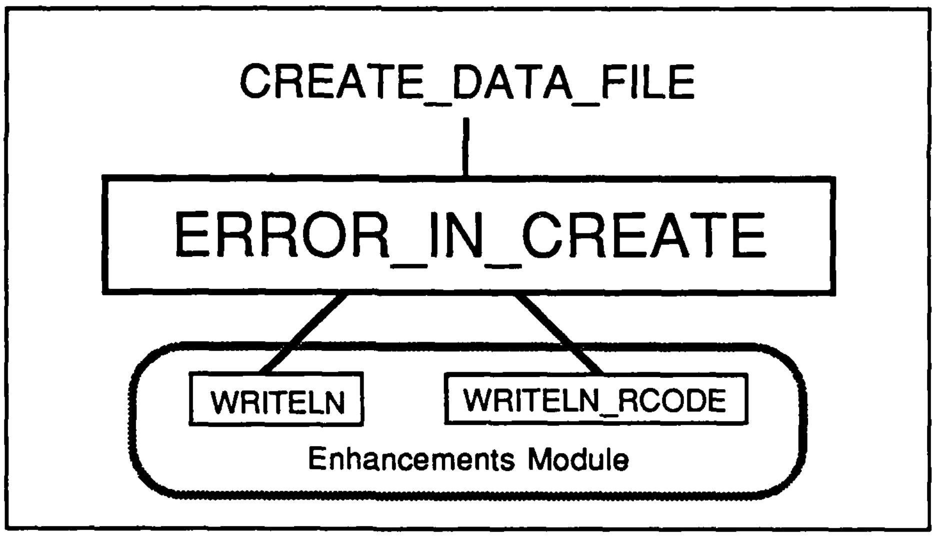


Figure 69. Relationship of ERROR_IN_CREATE to its Calling Routine and Subordinate Routines.

6. Invocation

ERROR_IN_CREATE is invoked from CREATE_DATA_FILE with

```
OUT_ERROR_CODE :=
        ERROR_IN_CREATE( IN_ERROR_CODE, RETURN_CODE )
```

where all three parameters are of type Byte.

7. Variables and Constants

ERROR_IN_CREATE uses no variables other than the input and output parameters. Several module level constants are used. Their names, values, and definitions are

Constant Name	Value	Definition
BAD_CHARACTER	BC hex	Error code for invalid character in a file name string. See VALID_STRING.
CONSOLE_OUT	Ø2 hex	The logical unit number for the monitor screen.
FATAL	FE hex	Error code for fatal error.
DUPLICATE_FILE	DØ hex	RIO return code for duplicate file name.

9. Other Routines Called

ERROR_IN_CREATE calls two of the output routines of Enhancements Module to write messages to the system console. The routines called and their invocations are

WRITELN(LOGICAL_UNIT, Pointer-To-Text-String)

WRITE_RCODE(LOGICAL_UNIT, RETURN_CODE)

where LOGICAL_UNIT (type Byte) is the number of the logical unit to be written to, Pointer-to-Text-String (type ASCII_PTR) points to the output text or is a constant text string, and RETURN_CODE (type Byte) is the completion code passed back from calls to the RIO Operating System. WRITLEN outputs a string of text followed by a carriage return to the designated logical unit. WRITE_RCODE outputs the text translation of the RIO return codes to the designated logical unit. WRITE_RCODE is used to output unexpected RIO return codes.

9. Output of Routine

ERROR_IN_CREATE writes messages and operating system return codes to the system console. In its current form, ERROR_IN_CREATE always returns the output parameter OUT_ERROR_CODE as FATAL.

10. Routine Flaws

ERROR_IN_CREATE would be improved by by using the IF statements within a DO loop structure like that used in PREPARE_COLLECTOR or the CASE statement structure like that used in ERROR_IN_PREPARE. Even if the structure isn't changed, ERROR_IN_CREATE needs to initially set ERROR_CODE to FALSE.

11. Reference to Listing

The program listing of ERROR_IN_CREATE is on page 382 in Appendix F.

1. Routine Name: **VALID_STRING**

2. Subordinate routine of Collect_Data Module.

3. Written in PLZ; 1Ø lines of code.

4. Synopsis of Routine

VALID_STRING checks the content of a text string passed to it to see whether it is a valid file name. Specifically, VALID_STRING ensures that each character in the string is a Ø through 9 or a A through Z. This check is accomplished by examining the ASCII value of each character against the ranges defined by the acceptable characters. Each character in the string is checked until an end of string is detected or 32 characters have been checked. If VALID_STRING finds any invalid charactes, the output ERROR_ CODE is set to BAD_CHARACTER. Otherwise ERROR_CODE is returned as FALSE, indicating no error.

5. Routine Relationship Diagram

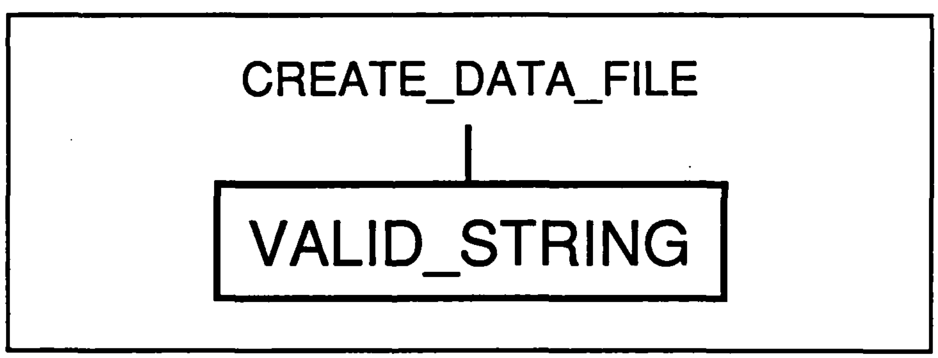


Figure 70. Relationship of VALID_STRING to CREATE_DATA_FILE.

6. Invocation

CREATE_DATA_FILE calls VALID_STRING with

ERROR_CODE := VALID_STRING(TEST_STRING)

where TEST_STRING is of type ASCII_PTR (for pointer to ASCII string) and ERROR_ CODE is of type Byte.

7. Variables and Constants

VALID_STRING uses one internal variable, INDEX (type Byte), in addition to the input and output variables. INDEX is used as a place keeper for the string TEST_STRING.

VALID_STRING uses two constants FALSE and BAD_CHARACTER. FALSE, value ØØ hex, is the error code for every thing is ok. BAD_CHARACTER, value BC hex, is the error code to signal that a invalid character was found.

8. Other Routines Called

VALID_STRING calls no other routines.

9. Output of Routine

VALID_STRING has two possible outputs. The output parameter OUT_ ERROR_CODE is either FALSE or BAD_CHARACTER. FALSE if no invalid characters were found in the string; BAD_CHARACTER if one invalid character was found.

10. Routine Flaws

VALID_STRING is acceptable, though its listing format and comments could be improved.

11. Reference to Listing

The listing of VALID_STRING's code is on page 383 in Appendix F.

1. Routine Name: **CREATE_DATA_FILE**

2. Primary subordinate routine of Collect_Data Module.

3. Written in PLZ; 29 lines of code.

4. Synopsis of Routine

CREATE_DATA_FILE is the third routine called by SAMPLE_DATA, the executive routine of Collect_Data Module. Using instructions passed into Collect_Data Module, CREATE_DATA_FILE opens a disk file into which the data collected by Sampler Module will be transfered. This requires the formation of a valid file name and a call to the operating system. As shown in the figure below, CREATE_ DATA_FILE calls many routines to accomplish these functions.

The file name formed has three fields separated by periods. The first field is the test identifier, passed into CREATE_DATA_FILE from the user. This field is six characters long and is susposed to be unique. Routine VALID_ STRING is called to ensure the user input has only valid file name parameters. If any of the characters are invalid, an error message is written to the console by ERROR_IN_CREATE and CREATE_DATA_FILE ends with ERROR_CODE being FATAL. The second field is the channel number. CREATE_DATA_FILE is pasbsed the input parameter INPUT_ CHANNEL, type Byte. The routine ASCII is called to translate INPUT_CHANNEL into the ASCII characters that are the base ten representation of INPUT_CHANNEL. The third field is the phrase "RAW_ DATA". Thus the file name looks like

testid . ## . RAW_DAT A

where "testid" is the unique test identifier and "##" are the characters that represent the input channel number.

With the file name formed, CREATE_DATA_FILE calls the operating system via OPEN, an external routine of the PLZ.STREAM.IO Module. If for any reason the opening is not successful, an error message is writen to the console by ERROR_IN_ CREATE, CREATE_DATA_FILE ends, and ERROR_CODE is returned as FATAL. If the opening is successful, CREATE_DATA_FILE proceeds.

With the data file open, CREATE_DATA_FILE continues by writing into the file the header information. Five extenal routines of the Enhancements Module are used by CREATE_DATA_FILE to write the header information to the disk file. The following is the content and format of the header.

testid:*testid*
| input_channel:*channel*
| peroid_value: *period_value*

| period_units: *period_units*
|#_samples: *samples*
|#_samples: *samples*
| date_of_test: *todays_date*
|user_message: *user_string*
|beginning of data:||

where the italized items are the names of the text string variables. Most of these text string variables are input parameters passed into CREATE_DATA_FILE. CHANNEL is formed in CREATE_DATA_FILE through the call to ASCII. The "|" character (ASCII C7 hex) is used as a field marker. With all the header information written into the data file, CREATE_DATA_FILE ends.

5. Routine Relationship Diagram

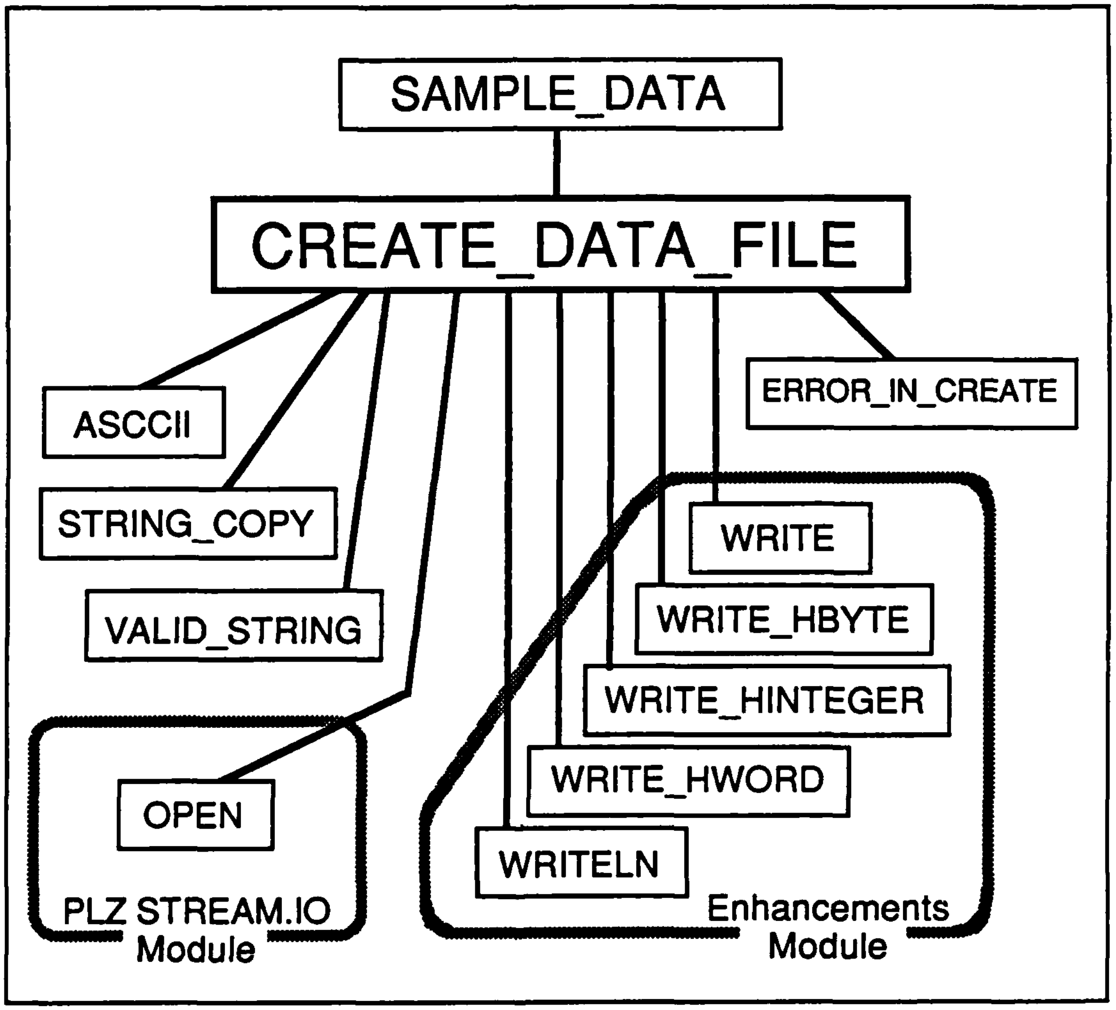


Figure 71. Relationships Between CREATE_DATA_FILE, SAMPLE_DATA, and Subordinate Routines.

6. Invocation

CREATE_DATA_FILE is invoked from SAMPEL_DATA with

```
ERROR_CODE := CREATE_DATA_FILE(  INPUT_CHANNEL,
                                 DATA_FILE,
                                 PERIOD_VALUE,
                                 PERIOD_UNITS,
                                 SAMPLES, TESTID,
                                 USER_MESSAGE,
                                 TODAYS_DATE     )
```

This routine has many input parameters, most of them passed in to become part of the data file header. Their types and purposes are

Parameter Name	Type	Definition
INPUT_CHANNEL	Byte	The number of the analog input channel, 1 to 16, data will be collected from.
DATA_FILE	Byte	The logical unit number for the file.
PERIOD_VALUE	Integer	The desired sampling interval (in PERIOD_UNITS units)
PERIOD_UNITS	Integer	The units of PERIOD_VALUE. The valid values are MICROSECONDS, MILLISECONDS, and SECONDS.
SAMPLES	Word	The number of samples to be collected.
TESTID	ASCII_PTR	A pointer to a six character (plus carriage return) string that is the unique test identifier.
USER_MESSAGE	ASCII_PTR	A pointer to a free field string of characters.
TODAYS_DATE	ASCII_PTR	A pointer to a six character string (plus a carriage return) that represent the date.

The single output parameter, ERROR_CODE (type Byte), passes back an error code to SAMPLE_DATA.

7. Variables and Constants

CREATE_DATA_FILE uses several internal variables in addition to the parameters discussed above.

Parameter Name	Type	Definition
FILE_NAME_BUF	ASCII_STRING	A 32 character buffer to hold the completed file name. 19 characters are used.
CHANNEL_BUF	ASCII_STRING	A 32 character buffer to hold the completed channel number. Only 3 characters are used.
FILE_NAME	ASCII_PTR	A pointer to FILE_NAME_BUF.
CHANNEL	ASCII_PTR	A pointer to CHANNEL_BUF
RETURN_CODE	Byte	Receives the operating system return code from the call to OPEN.

In addition to these variables, CREATE_DATA_FILE uses two constants, OPERATION_COMPLETE (value 8Ø hex) and FATAL (value FE hex). OPERATION_ COMPLETE is the RIO Operating System return code for all went well. FATAL is the Colect_Data Module error code that signals fatal errors.

8. Other Routines Called

As was shown in the figure above, CREATE_DATA_FILE calls ten routines. Their invocations and parameters follow. Unless otherwise stated, the routines are part of Collect_Data Module.

a. TEXT_STRING := ASCII(NUMBER, INDEX, DIVISOR, INPOINTER)

where TEXT_STRING and INPOINTER are type ASCII_PTR, and NUMBER, INDEX, and DIVISOR are type Word. ASCII converts NUMBER into the string of ASCII characters which represent it.

b. STRING_COPY(SOURCE, S_INDEX, DESTINATION, D_INDEX)

where SOURCE and DESTINATION are type ASCII_PTR, and S_INDEX and D_INDEX are type Byte. STRING_COPY transcribes the characters of SOURCE string into the DESTINATION string.

c. ERROR_CODE := VALID_STRING(TEST_STRING)

where ERROR_CODE is type Byte and TEST_STRING is type ASCII_PTR. VALID_ STRING ensures the characters in TEST_STRING are valid for inclusion in a file name.

d. RETURN_CODE := OPEN(LOGICAL_UNIT, FILE_NAME_PTR, MODE)

where RETURN_CODE, LOGICAL_UNIT, and MODE are type Byte and FILE_ NAME_ PTR is type PByte for pointer to byte. OPEN is an external routine of the PLZ.STREAM.IO Module. OPEN calls the operating system to open a disk file.

e. WRITE(LOGICAL_UNIT, TEXT_POINTER)

where LOGICAL_UNIT is type Byte and TEXT_POINTER is type PByte. WRITE is an external routine of the Enhancements Module. WRITE outputs the text pointed to by TEXT_POINTER to the desired LOGICAL_UNIT.

f. WRITE_HBYTE(LOGICAL_UNIT, VALUE)

where both parameters are type Byte. WRITE_HBYTE is an external routine of the Enhancements Module. WRITE_HBYTE outputs the two ASCII characters that represent the VALUE.

g. WRITE_HINTEGER(LOGICAL_UNIT, VALUE)

where LOGICAL_UNIT is type Byte and VALUE is type Integer. WRITE_HINTEGER is an external routine of the Enhancements Module that outputs the characters which form the hexidecimal representatio of VALUE to the designated LOGICAL_UNIT.

h. WRITE_HWORD(LOGICAL_UNIT, VALUE)

where LOGICAL_UNIT is type Byte and VALUE is type Word. WRITE_HWORD is an external routine of the Enhancements Module. WRITE_HWORD outputs the four characters which form the hexidecimal representation of VALUE to the designated LOGICAL_UNIT.

i. WRITELN(LOGICAL_UNIT, TEXT_POINTER)

where LOGICAL_UNIT is type Byte and TEXT_POINTER is type PByte. WRITELN is an external routine of the Enhancements Module. WRITELN, like WRITE above, outputs text; WRITELN adds a carriage return at the end of the text string.

j. OUT_ERROR_CODE := ERROR_IN_CREATE(IN_ERROR_CODE,
RETURN_CODE)

where all three parameters are type Byte.

Please consult the descriptions of these routines for more information on their function.

9. Output of Routine

If something goes wrong during the execution of CREATE_DATA_FILE, the output of the routine is ERROR_CODE filled with FATAL. If all goes well, the output of CREATE_DATA_FILE is an open disk file with the header information written in. The output parameter ERROR_CODE will hold FALSE indicating successful operation.

10. Routine Flaws

The major omission in CREATE_DATA_FILE is that VALID_STRING isn't called to check the TESTID or the CHANNEL number. Both VALID_STRING and ERROR_IN_CREATE calls should follow the STRING_COPY calls. The second problem in CREATE_DATA_FILE is that STRING_COPY is improperly called COPY_STING. The routine also badly needs commenting.

11. Reference to Listing

CREATE_DATA_FILE's listing is on page 384-385 in Appendix F.

1. Routine Name: **LOAD_DATA_FILE**

2. Primary subordinate routine of Collect_Data Module.

3. Written in PLZ; 16 lines of code.

4. Synopsis of Routine

LOAD_DATA_FILE reads the data loaded into memory by SAMPLER and loads that data into the disk file opened by CREATE_DATA_FILE. Were it not for error checking, LOAD_DATA_FILE would simply be a call to PUTSEQ, an external routine of the PLZ.STREAM.IO Module, to write the data to memory. Two error checks are present however. First, a check is made after the call to PUTSEQ checking that the number of bytes that should have been written to disk were written to disk. If the numbers don't match, an error message is written to the system console via the external output routines of the Enhancements Module. The output parameter ERROR_CODE is set to STORAGE_ERROR. The second error check is on the operating system return code from the PUTSEQ call. If the code is no OPERATION_COMPLETE an error message is again written to the console. In this case, the returned ERROR_CODE is FATAL. The figure below shows the relationship between LOAD_DATA_FILE and the external routines.

5. Routine Relationship Diagram

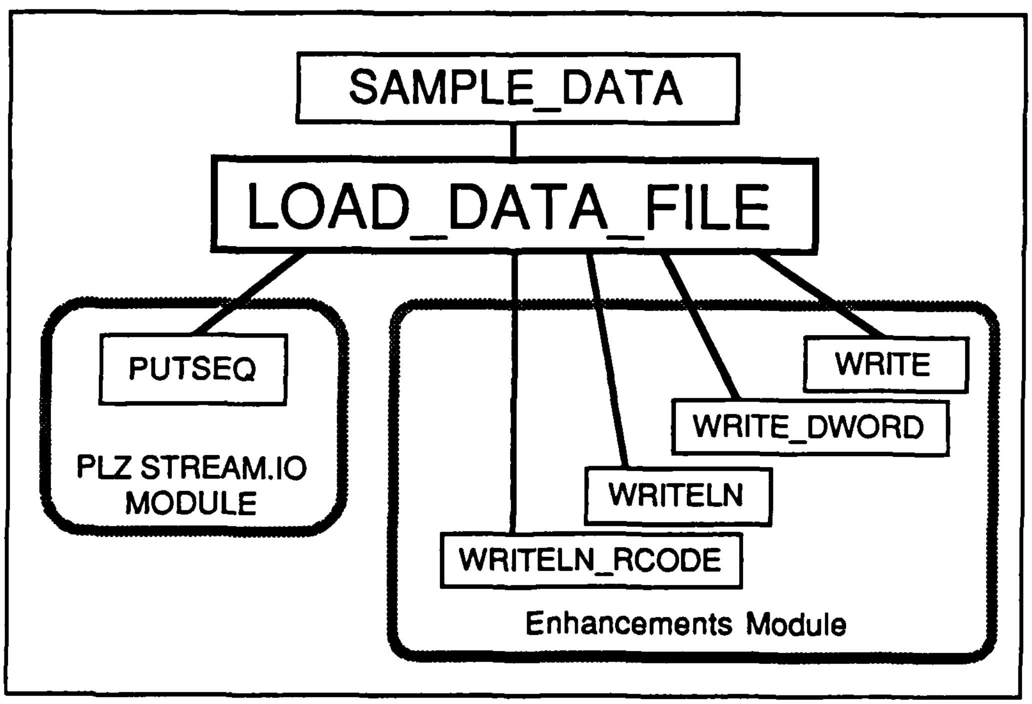


Figure 72. Relationship of LOAD_DATA_FILE to Other Routines.

6. Invocation

LOAD_DATA_FILE is invoked from SAMPLE_DATA by

```
ERROR_CODE := LOAD_DATA_FILE(   DATA_FILE,
                                BUFFER_BEGINNING,
                                LAST_DATA            )
```

where ERROR_CODE and DATA_FILE are type Byte and BUFFER_BEGINNING and LAST_DATA are type PByte for Pointer to Byte.

7. Variables and Constants

LOAD_DATA_FILE uses three internal variables. Their types and purposes are listed below.

Variable Name	Type	Definition
NUMBER_OF_BYTES	Word	The number of bytes of data in the memory buffer
BYTES_WRITTEN	Word	Receives the return parameter from PUTSEQ that says how many bytes were actually output.
RETURN_CODE	Byte	Receives the return parameter from PUTSEQ that holds the operating system return code.

LOAD_DATA_FILE also uses a few constants. They are

Constant Name	Type	Definition
STORAGE_ERROR	23 hex	Error code for mismatch in number of bytes written vs number of bytes in memory buffer.
CONSOLE_OUT	Ø2 hex	Logical unit number for the system console.

Constant Name	Type	Definition
OPERATION_COMPLETE	8Ø hex	RIO return code for successful operation.
FATAL	FE hex	Error code for a fatal error in Collect_Data.

8. Other Routines Called

LOAD_DATA_FILE calls five external routines. Their invocations, parameters, and functions are listed below.

a. RETURN_CODE := PUTSEQ(LOGICAL_UNIT,
BUFFER_PTR,
NUMBER_OF_BYTES)

where RETURN_CODE and LOGICAL_UNIT are type Byte, BUFFER_PTR is type pointer to Byte, and NUMBER_OF_BYTES is type Word. This external routine of the PLZ.STREAM.IO Module is used by LOAD_DATA_FILE to write the data stored in memory into the disk file.

b. WRITE(LOGICAL_UNIT, TEXT_POINTER)

where LOGICAL_UNIT is type Byte and TEXT_POINTER is type PByte for pointer to byte. LOAD_DATA_FILE uses WRITE to output error messages to the system console. WRITE is part of the Enhancements Module.

c. WRITE_DWORD(LOGICAL_UNIT, VALUE)

where LOGICAL_UNIT is type Byte and VALUE is type Word. This external routine of the Enhancements Module is used to output decimal values to the system console.

d. WRITELN(LOGICAL_UNIT, TEXT_POINTER)

where LOGICAL_UNIT is type Byte and TEXT_POINTER is type PByte. LOAD_DATA_ FILE uses this Enhancements Module routine to output strings of characters to the system console. WRITELN, unlike WRITE, outputs a carriage return at the end of the character string.

e. WRITELN_RCODE(LOGICAL_UNIT, RETURN_CODE)

where both LOGICAL_UNIT and RETURN_CODE are type Byte. LOAD_DATA_FILE uses WRITELN_RCODE to translate the operating system return code into text and then output the text to the system console. WRITELN_RCODE is an external routine, an element of the PLZ.STREAM.IO Module.

9. Output of Routine

If all goes well in LOAD_DATA_FILE, the result will be a data file loaded with the data from the memory buffer and an ERROR_CODE of FALSE. If things don't go well, error messages will be written to the console, the data file will be in an indeterminant state, and the ERROR_CODE will be FATAL or STORAGE_ERROR.

10. Routine Flaws

LOAD_DATA_FILE is flawed in that ERROR_CODE is in an indeterminant state if every thing goes well. To fix this flaw, an additional statement initializing ERROR_ CODE to FALSE (error code for no error) should be added to LOAD_DATA_FILE. This statement should be inserted prior to the PUTSEQ call. Also, LOAD_DATA_FILE is devoid of commenting.

11. Reference to Listing

The program listing of LOAD_DATA_FILE is on page 386 in Appendix F.

1. Routine Name: **CLOSE_DATA_FILE**

2. Primary subordinate routine of Collect_Data Module

3. Written in PLZ; 7 lines of code.

4. Synopsis of Routine

This short routine closes the data file opened by CREATE_DATA_FILE and filled by LOAD_DATA_FILE; it is the last routine called by SAMPLE_DATA, the executive routine of Collect_Data Module. CLOSE_DATA_FILE closes the file with a call to the external routine CLOSE. If the operation was successful, CLOSE_DATA_FILE ends. Otherwise, CLOSE_DATA_FILE outputs an error message to the sytem console and returns the FATAL error code. The relationship of CLOSE_DATA_FILE to its calling routine and its subordinate routines is shown in the figure below.

5. Routine Relationship Diagram

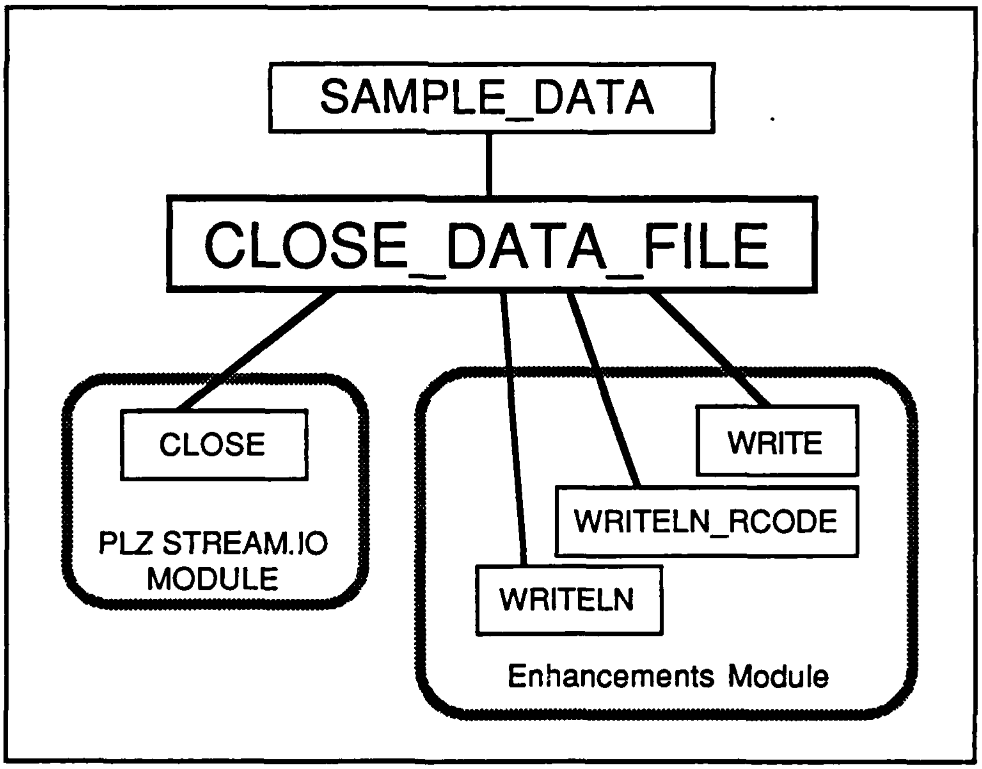


Figure 73. Relationship of CLOSE_DATA_FILE to Other Routines

6. Invocation

CLOSE_DATA_FILE is invoked from SAMPLE_DATA via

ERROR_CODE := CLOSE_DATA_FILE(DATA_FILE)

where both ERROR_CODE and DATA_FILE are type Byte. ERROR_CODE returns to SAMPLE_DATA a code indicating success or failure. DATA_FILE is the logical unit number of the data file.

7. Variables and Constants

CLOSE_DATA_FILE uses one internal variable, RETURN_CODE (type Byte). This variable receives the return parameter from the call of CLOSE.

Two constants are used by CLOSE_DATA_FILE, OPERATION_COMPLETE (value 8Ø hex) and FATAL (value FE hex). OPERATION_COMPLETE is the RIO Operating System return code for successful completion. FATAL is the Collect_Data Module error code for failed operations.

8. Other Routines Called

LOAD_DATA_FILE calls four external routines. Their names, parameters, and functions are listed below.

RETURN_CODE := CLOSE(LOGICAL_UNIT)

where both parameters are type Byte. RETURN_CODE is the operating system's message on success or failure of the file closing procedure. LOGICAL_UNIT is the number of the unit to be closed. CLOSE is a routine of the PLZ.STREAM.IO Module.

WRITE(LOGICAL_UNIT, TEXT_STRING)

where LOGICAL_UNIT (type Byte) is the device number to which output is directed and TEXT_STRING (type PByte) is a pointer to the string of text to be output. CLOSE_DATA_ FILE uses WRITE to output an error message to the system console. WRITE is a member of the Enhancements Module.

WRITELN_RCODE(LOGICAL_UNIT, RETURN_CODE)

where both parameters are of type Byte. LOGICAL_UNIT is the device number to which output is directed. RETURN_CODE is the operating system code that WRITELN_RCODE will translate into its text definition and output the text to the designated LOGICAL_UNIT. CLOSE_DATA_FILE uses this external routine from the Enhancements Module to output the translated return code to the console in the error message.

WRITELN(LOGICAL_UNIT, TEXT_POINTER)

where LOGICAL_UNIT is type Byte and TEXT_POINTER is type PByte. LOGICAL_UNIT is the device number for the output. TEXT_POINTER points to the string of characters to be output. WRITELN, like WRITE, is used by CLOSE_DATA_FILE to send error messages to the system console. Unlike WRITE, WRITELN concludes the text string with a carriage return. WRITELN is an external routine from Enhancements Module.

9. Output of Routine

CLOSE_DATA_FILE closes the disk file into which the data collected from the AIO board was stored.

10. Routine Flaws

Like several other routines of Collect_Data Module, ERROR_CODE is not initialized. An additional line of code to initialize ERROR_CODE to FALSE is needed. Also like the later routines of Collect_Data Module, CLOSE_DATA_FILE needs commenting.

11. Reference to Listing

CLOSE_DATA_FILE's code listing is on page 387 in Appendix F.

1. Routine Name: **ERROR_IN_SAMPLER**

2. Subordinate routine of Collect_Data Module

3. Written in PLZ; 8 lines of code.

4. Synopsis of Routine

ERROR_IN_SAMPLER is a error detection / error message writing routine that SAMPLE_DATA calls after the call to the external routine SAMPLER. SAMPLER returns an error code to SAMPLE_DATA. If the error code is other than FALSE, SAMPLE_DATA calls ERROR_IN_SAMPLER to send the proper error messages to the system console. ERROR_IN_SAMPLER also calls CLOSE_DATA_FILE and returns to SAMPLE_DATA a FATAL error code.

5. Routine Relationship Diagram

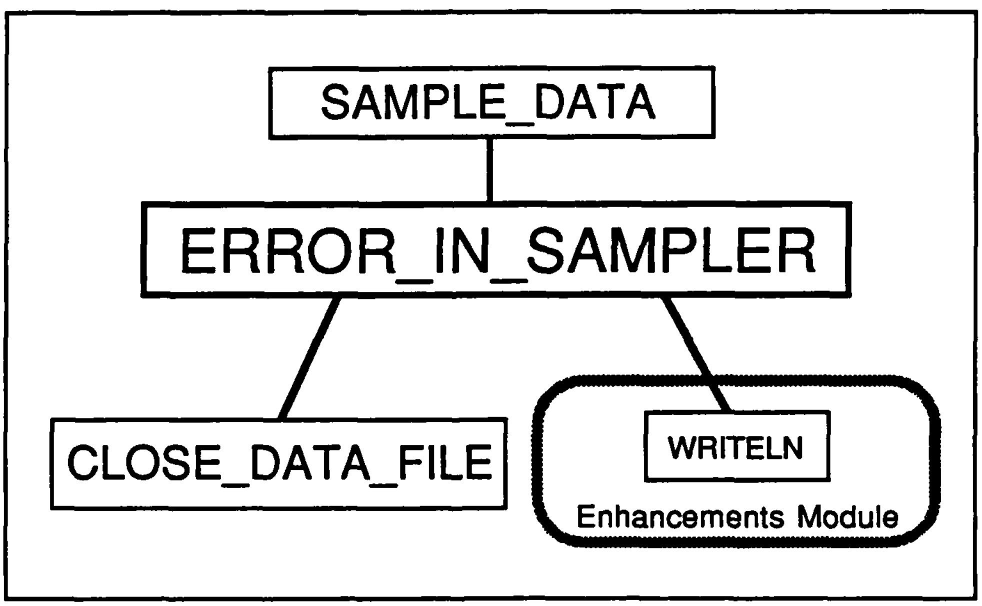


Figure 74. Relationship of ERROR_IN_SAMPLER to SAMPLE_DATA, CLOSE_DATA_FILE, and WRITELN

6. Invocation

The invocation of ERROR_IN_SAMPLER from SAMPLE_DATA is

OUT_ERROR_CODE := ERROR_IN_SAMPLER(IN_ERROR_CODE)

where both parameters are of type Byte.

7. Variables and Constants

No variables other than the input and output parameters are used by ERROR_IN_SAMPLER. Four constants are used. ABORT (value AB hex) is the error code from SAMPLER that data collection was terminated. CONSOLE_OUT (value Ø2 hex) is the logical unit number of the system console. DATA_FILE (value Ø7 hex) is the logical unit number of the disk file opened by CREATE_DATA_FILE. Last is FATAL (value FE hex), the error code for a fatal error.

8. Other Routines Called

ERROR_IN_SAMPLER calls two routines, WRITELN and CLOSE_DATA_FILE. Their invocations, parameters, and functions follow.

WRITELN(LOGICAL_UNIT, TEXT_POINTER)

where LOGICAL_UNIT (type Byte) indicates the logical unit and TEXT_POINTER (type PByte for Pointer to Byte) points to the string to be output. This external routine of the Enhancements Module also outputs a carriage return at the end of the text string. ERROR_IN_SAMPLER uses WRITELN to output error messages to the system console.

ERROR_CODE := CLOSE_DATA_FILE(FILE_UNIT)

where ERROR_CODE (type Byte) indicates whether the closing was successful and FILE_UNIT (type Byte) is the logical unit number of the file to be closed.

9. Output of Routine

ERROR_IN_SAMPLER is called only if SAMPLE_DATA finds a non-FALSE error code returning from SAMPLER. Thus something has already gone wrong. ERROR_IN_SAMPLER's output is messages to the system console and the closing of the data file opened by CREATE_DATA_FILE. ERROR_IN_SAMPLER always returns a FATAL error code.

10. Routine Flaws

The only flaw is that SAMPLE_DATA calls ERROR_IN_SAMPLER only when it detects an error. A superior organization would be to have SAMPLE_DATA call ERROR_IN_SAMPLER immediately after SAMPLER without checking the error code. ERROR_IN_SAMPLER would determine if any error had occured and return an error code of FALSE for all nonfatal errors. The IF statements inside a DO loop structure used by PREPARE_COLLECTOR would be one approach with IF statements for each expected error code and a "wild error" message for the unexpected.

11. Reference to Listing

The program listing of ERROR_IN_SAMPLER is on page 388 in Appendix F.

1. Routine Name: **SAMPLE_DATA**

2. Executive routine of Collect_Data Module.

3. Written in PLZ; 17 lines of code.

4. Synopsis of Routine

SAMPLE_DATA is the executive routine of the Collect_Data Module. All the other routines of Collect_Data Module are called either directly or indirectly by SAMPLE_DATA. The figure below shows the basic execution flow of SAMPLE_DATA and the principal subordinate routines it calls. Included in this list is SAMPLER, the executive routine of Sampler Module, the assembly language module that performs the actual data collection.

The execution revolves around five major processes. First User supplied instructions are translated by PREPARE_COLLECTOR into the command necessary to program the CTC driven interrupt timer. Next, again with user inputs, a disk file is opened in process CREATE_DATA_FILE. Third, the analog data is read in and stored in memory. This process is the responsibility of the Sampler Module. The data stored in memory is then written into the disk file by LOAD_DATA_FILE. Lastly, the now filled data file is closed by CLOSE_DATA_FILE. Thoughout this process, if anything goes wrong an error message is output to the system console.

SAMPLE_DATA, and hence all of Colllect_Data Module, is designed to be called from a superior PLZ routine. The responsibility of that superior routine is the interface with the user.

5. Routine Relationship Diagram

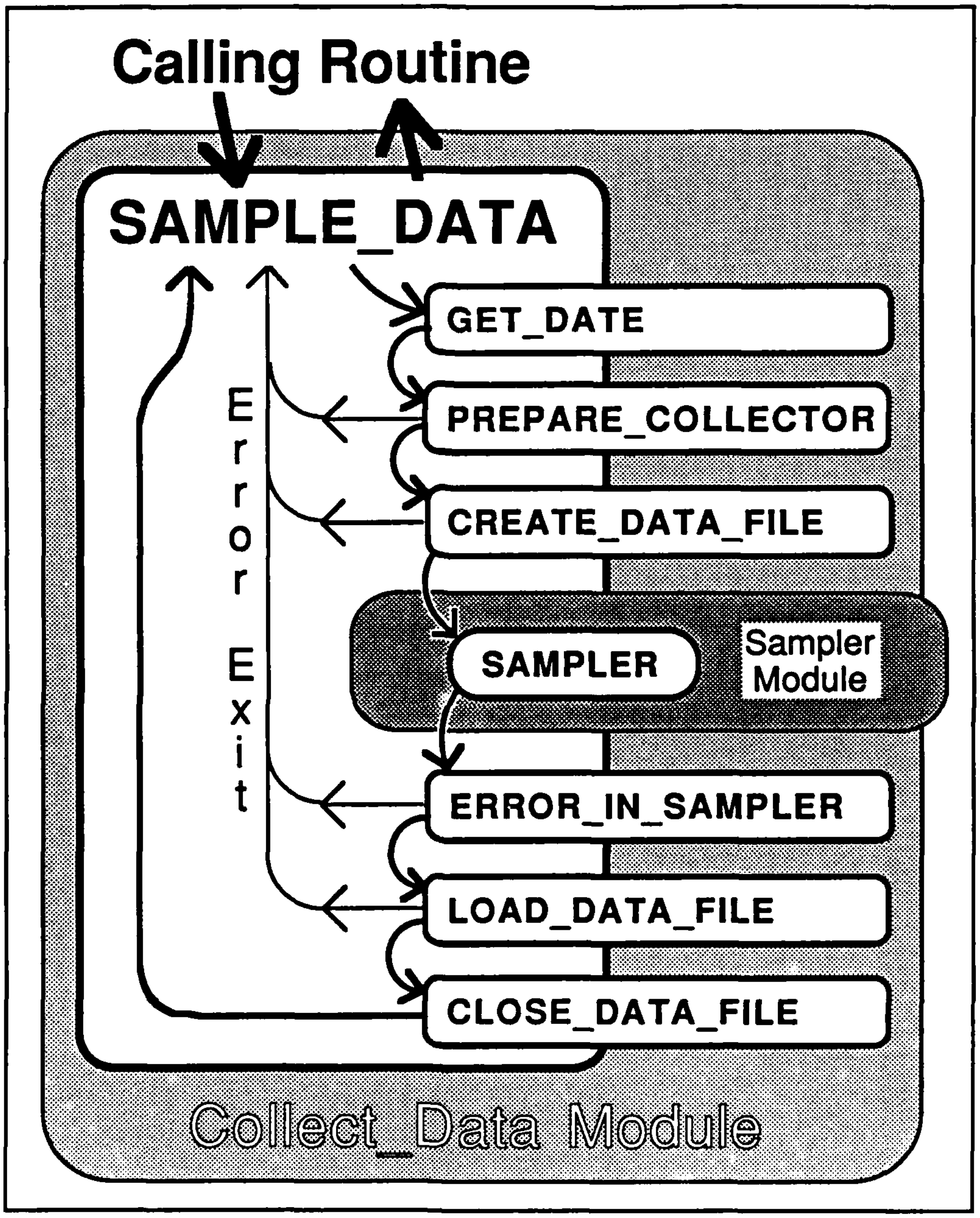


Figure 75. Relationship of SAMPLE_DATA to its Calling Routine and to its Subordinate Routines.

6. Invocation

SAMPLE_DATA is invoked from its calling routine with

ERROR_CODE := SAMPLE_DATA(TESTID, USER_MESSAGE, PERIOD_VALUE, PERIOD_UNITS, INPUT_CHANNEL, SAMPLES)

The type and purpose of these parameters is listed below.

Parameter Name	Type	Definition
ERROR_CODE	Byte	Return parameter to indicate to the calling routine whether execution was successful.
TESTID	ASCII_STRING	A string holding the six character sequence that uniquely identifies the test.
USER_MESSAGE	ASCII_STRING	A string holding a free field message.
PERIOD_VALUE	Integer	The number of time units desired for the sampling period.
PERIOD_UNITS	Integer	The units of PERIOD_VALUE. Defined units are MICROSECONDS, MILLI-SECONDS, and SECONDS.
INPUT_CHANNEL	Byte	The number of the desired analog input channel on the AIO board.
SAMPLES	Word	The number of data samples the user wants collected.

7. Variables and Constants

SAMPLE_DATA uses three variables in addition to the parameters addressed above. They are

Variable Name	Type	Definition
TODAYS_DATA_BUF	ASCII_STRING	A buffer to hold the characters that represent the date (yymmdd)

Variable Name	Type	Definition
TODAYS_DATE	ASCII_PT	A pointer to TODAYS_DATE_BUF.
LAST_DATA	PByte	A pointer to the memory location that holds the last byte of the data collected.

SAMPLE_DATA also makes use of a number of constants. They are

Constant Name	Type	Definition
FALSE	ØØ hex	The error code for nothing went wrong.
FATAL	FE hex	The error code a fatal error.
DATA_FILE	Ø7 hex	The logical unit number for the data file.

8. Other Routines Called

SAMPLE_DATA calls seven subordinate routines and uses one buffer. Six of these are members of the Collect_Data Module. They are

GET_DATE,
PREPARE_COLLECTOR,
CREATE_DATA_FILE,
ERROR_IN_SAMPLER,
LOAD_DATA_FILE, and
CLOSE_DATA_FILE.

The invocation, parameters, and functions of these routines will not be discussed here. These items are detailed in the descriptions of these routines. One important note however. Several of the above routines call input/output routines of the PLZ.STREAM.IO Module. The PLZ.STREAM.IO Module must be linked in with Collect_Data Module. The routines called, OPEN, CLOSE, and PUTSEQ, must be declaired external routines.

The other subordinate routine called by SAMPLE_DATA is SAMPLER, an external routine of the Sampler Module. SAMPLER sets up the CTC interrupts, programs the AIO analog to digital input, polls the user for a "GO" signal, and then reads in data from the AIO board and stores it in memory. SAMPLER is invoked from SAMPLE_DATA with

```
ERROR_CODE, LAST_DATA :=
                    SAMPLER(  IO_CHANNEL, CTC_MODE,
                              TIME_CNST, COUNT,
                              SAMPLES, FIRST_DATA     )
```

The type and purpose of these input and output parameters follows.

Parameter Name	Type	Definition
ERROR_CODE	Byte	Returns a code which indicates whether data collection was successful or tells what went wrong. Five codes are defined. FALSE: no error; ABORT: user abort; FATAL: complete breakdown; CHANNEL_INVALID: channel number was out of range; MODE_ INVALID CTC commands were invalid.
LAST_DATA	PByte	Returns a pointer to the memory location in which the last byte of data was stored.
IO_CHANNEL	Byte	Passes in the number of the AIO Channel to be used for data collection. (Ø to 15)
CTC_MODE	Byte	The first of two programming bytes for the CTC passed into SAMPLER. CTC_MODE has two possible values, FAST_MODE (87 hex) and SLOW_ MODE (A7 hex). Both set the CTC to generate periodic interrupts.
TIME_CNST	Byte	The second CTC timing value passed in. Its defined range is Ø to 255 decimal. This byte tells the CTC the value of the internal counter.
COUNT	Word	The number of CTC generated interrutps per AIO analog to digital conversion.
NUM_SAMPLES	Word	The number of twelve bit samples (stored in two eight bit locations) to be collected.

Parameter Name	Type	Definition
FIRST_DATA	PByte	A pointer to the memory location where the first byte of data collected is to be stored.

In order for SAMPLER to be called by SAMPLE_DATA, it must be declaired external and Sampler Module must be linked in with Collect_Data Module.

The third external structure used by SAMPLE_DATA is DATA_BUFFER, a 2,ØØØ byte memory allocation set up by Buffers Module. This buffer is used inconjunction with routine SIZE_DATA_BUFFER (an internal routine subordinate to PREPARE_COLLECTOR) to define the storage space used by SAMPLER to store the data read in from the AIO board. This simple approach is only indended for initial checkout of Collect Data Module. Ultimately, SIZE_DATA_BUFFER would work directly with the RIO operating system memory manager. Then, the data buffer would be dynamically allocated rather than limited to some arbitrary preselected size. Access to the memory manager from SIZE_DATA_BUFFER would be provide by ALLOCATE and DEALLOCATE, external routines of the Utility Module.

9. Output of Routine

There are two classes of SAMPLE_DATA (and hence Collect_Data) outputs. First, if all goes sufficiently well in both Collect_Data Module and Sampler Module, the output will be a new file on the system. That file will contain header information on the data collected and up to 2,ØØØ bytes of data read in from the AIO analog to digital converter. The second class of outputs covers the outcome when fatal errors or user aborts occur. For most errors, execution of SAMPLE_DATA will end. For a couple of cases, a data file will have been created and filled with header information but no data will be present.

10. Routine Flaws

There is one structural flaw in SAMPLE_DATA , a number of listing errors, and a lack of comments. The sturctural flaw is that if a fatal error occurs during LOAD_ DATA_FILE, the current logic flow has SAMPLE_DATA just end, leaving the data file open. Reguardless of the error, the data file should be closed by calling CLOSE_ DATA_FILE. The conditional exit following the call to LOAD_DATA_FILE should be eliminated.

The listing of SAMPLE_DATA has three mislabled parameters in calls to subordinate routines. First, in the call to CREATE_DATA_FILE , the third input parameter should be the constant DATA_FILE rather than FILE_UNIT. Then in the calls to SAMPLER and LOAD_DATA_FILE, the parameter currently listed as BEGINNING_OF_ BUFFER should either be changed to ^DATA_BUFFER[Ø] (a pointer to the first location of the data buffer) or be defined as local variable of type PByte and set equal to ^DATA_ BUFFER[Ø] early in SAMPLE_DATA.

The final flaw in SAMPLE_DATA is its lack of comments. The use of an alternate format for the rather lengthy subordinate routine calls would also aid readability of the routine.

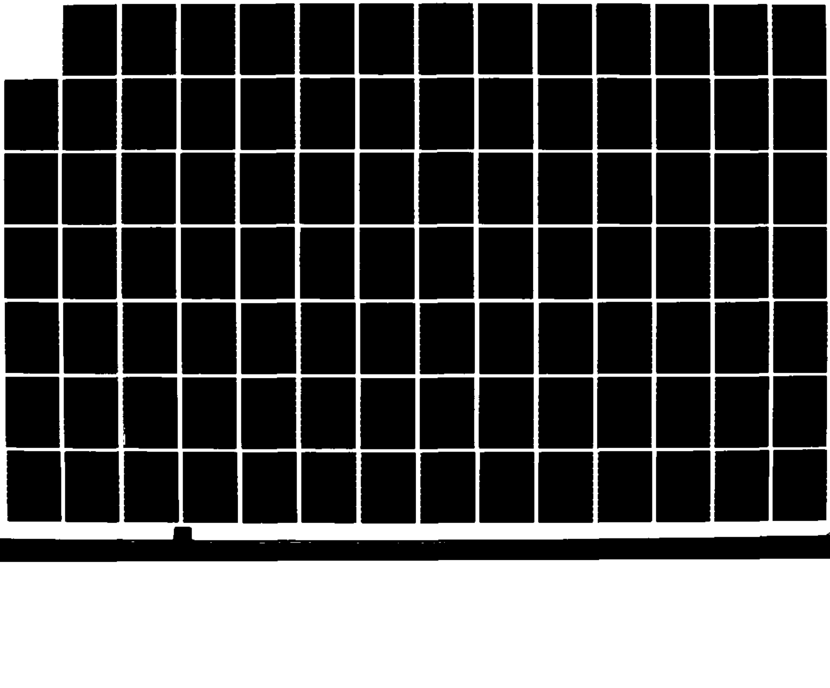
11. Reference to Listing

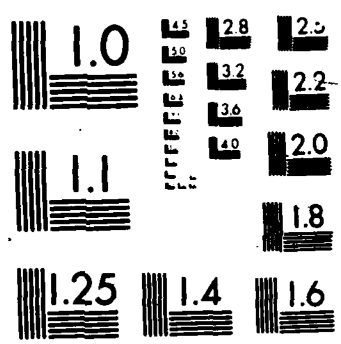
The listing of SAMPLE_DATA is on page 389 in Appendix F.

This page is intentionally blank.

AD-A172 823 DESIGN AND PARTIAL IMPLEMENTATION OF A COMPUTER CONTROLLED DATA COLLECTION SYSTEM(U) AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH SCHOOL OF ENGI.. L E LUTZ 4/5

UNCLASSIFIED FEB 86 AFIT/GE/ENG/86M-1 F/G 9/2 NL




1.0
2.8
2.5
3.2
2.2
3.6
4.0
2.0
1.1
1.8
1.25
1.4
1.6

VII. Conclusion

The system designed around the hardware of the Zilog MCZ Z-8Ø Development System is a reasonable, general purpose data collection system. The design supports the requirements for acuracy, data integrity, flexibility, and a simple user interface presented in Section 1. The design is based on having an data storage unit located with in the item under test. This internal unit would store the data until post test when the data would be transfered out to an external control and data storage unit.

The purpose of the thesis effort was to examine and develop the software required to implement such a data collection system. The first step was to provide some improvements to the PLZ language. The software written to improve the Pascal-like PLZ language proved quite useful and effective. These IO routines, written in PLZ, of the Enhancements Module were fully developed and tested. Hardware and operating system access routines were also written to supplement PLZ. These assembly language routines of the Utility Module were also fully developed and checked out.

Like the PLZ improvement software, the software written for the data collection system was written in both PLZ and Z-8Ø assembly language. Since a single development system was used for both the internal data collection / temporary storage unit and the external control / archive unit, the division of software between the units became some what blurred. The software of the Collect_Data Module, and its subordinate Sampler Module, implement one of the five processes of the data collection system, the collection and storage of data. The Sampler Module software is that of the internal data collection / temporary storage unit; the Collect_Data software would be resident in the external control / data archive unit. This software was never fully functional. The problem appears to be in the interface between PLZ language calling routines and the Z-8Ø assembly language Sampler Module. Though the software was not functional, it did fulfull the purpose of examining the software required to implement a data collection system.

Recommendations

Two courses of future action are clearly open. The software of the data collection system could be completed and thoroughly tested. This would include integration of the Set Up Scale Factor File software and implementation of the three processes (Scale Data, Output Data, and User Data Manipulations) not implemented during this effort. The second course of action would be to build one of the internal data collection units. This activity should not be started until the software portion of the system is complete.

Bibliography

1. Zilog. Z-8Ø-CPU, Z-8ØA-CPU Product Specification. Zilog, Inc. 1Ø46Ø Bubb Road, Cupertino, California 95Ø14, March 1978.

2. Zilog. Z8Ø-MCB Hardware User's Manual. Revision A. Zilog, Inc. 1Ø46Ø Bubb Road, Cupertino, California 95Ø14, 8 May 1978.

4. Zilog. Z-8Ø MCB Software User's Manual. Revision H. Zilog, Inc. 1Ø46Ø Bubb Road, Cupertino, California 95Ø14, July 1979.

5. Zilog. Report on the Programming Language PLZ/SYS. Zilog, Inc. 1Ø46Ø Bubb Road, Cupertino, California 95Ø14, undated.

6. Zilog. PLZ version 3 User Guide. Revision H. Zilog, Inc. 1Ø46Ø Bubb Road, Cupertino, California 95Ø14, July 1979.

7. *Zilog. Z8Ø SIB User's Manual. Revision B. Zilog, Inc.* 1Ø46Ø Bubb Road, Cupertino, California 95Ø14, 28 July 1978.

8. Zilog. Z8Ø-AIO/AIB Hardware User's Manual. Revision A. Zilog, Inc. 1Ø46Ø Bubb Road, Cupertino, California 95Ø14, 28 April 1978.

9. Barden, William Jr. The Z-8Ø Microcomputer Handbook. Indianapolis, Indiana: Howard W. Sams & Co., Inc., 1978.

10. Grogono, Peter. Programming in Pascal. Reading, Massachusetts: Addison-Wesley Publishing Company, Inc., 1978.

11. Cave, Stephen. Telephone Interview, 3 April 1980. EG&G Corporation, Albuquerque, NM.

12. Aeby, Charles A., Project Officer. Telephone Interview, 5 May 1980, Air Force Weapons Laboratory, Kirtland AFB, NM.

Appendix A: Enhancements Module Listings

The following 36 pages are the compiler listing of the Enhancements Module, the DEBUGS Module, and TEST_IT Module. DEBUGS Module is a special subset of Enhancements Module used for debugging of PLZ programs which interact with the RIO Operating System. TEST_IT Module is one set of routines used to test the routines of Enhancements Module. The following is a list of the routines found on each page.

PLZSYS 3.0 810211.1417

```
ENHANCEMENTS MODULE             ! 1328 - 11 February 1981  !

    !     This PLZ module contains many procedures for input and ouput of   !
    ! BYTE, INTEGER, WORD, and text strings to any logical units.  These    !
    ! routines have been designed to emulate, as much as possible, the PACAL !
    ! WRITE, WRITELN, READ, and READLN functions.                            !

  CONSTANT
    BELL            := %07   ! Aural signal - control G  !
    BACK_SPACE      := %08   ! Control H                 !
    TAB             := %09   ! Horizonal Tab - control I !
    LINE_FEED       := %0A   ! Control J                 !
    CARRIAGE_RETURN := %0D   ! Control M                 !
    ESCAPE          := %1B   ! Control [                 !
    BLANK           := %20
    TRUE  := 1
    FALSE := 0
    OPERATION_OK    := %80   ! RIO return code for successful operation. !


  TYPE

    PBYTE ^BYTE
    ASCII_STR ARRAY [ 8 BYTE ]
    ASCII_PTR ^ASCII_STR


  EXTERNAL

    PUTSEQ PROCEDURE( LOGICAL_UNIT BYTE, BUFFER_PTR PBYTE, NUMBER_BYTES WORD )
           RETURNS  ( RETURN_BYTES WORD, RETURN_CODE BYTE )

    GETSEQ PROCEDURE( LOGICAL_UNIT BYTE, BUFFER_PTR PBYTE, NUMBER_BYTES WORD )
           RETURNS  ( RETURN_BYTES WORD, RETURN_CODE BYTE )
```

```
        INTERNAL    !!!!!!!!!!! routines necessary for operation of global routines !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!


         ASCII PROCEDURE( VALUE WORD,  IN_BLANKING BYTE )
                RETURNS( CHARACTER   OUT_BLANKING BYTE )

         ! This routine takes the input value (0 to F hex) and returns the ASCII !
         ! character which represents this value.  If IN_BLANKING is passed in   !
         ! as TRUE, CHARACTER will be returned as a blank.                       !

          ENTRY
           OUT_BLANKING := TRUE
           IF ( VALUE = %0 ) ANDIF ( IN_BLANKING = TRUE )
            THEN CHARACTER := BLANK
            ELSE
             OUT_BLANKING := FALSE
             IF VALUE
              CASE %0 THEN  CHARACTER := '0'
              CASE %1 THEN  CHARACTER := '1'
              CASE %2 THEN  CHARACTER := '2'
              CASE %3 THEN  CHARACTER := '3'
              CASE %4 THEN  CHARACTER := '4'
              CASE %5 THEN  CHARACTER := '5'
              CASE %6 THEN  CHARACTER := '6'
              CASE %7 THEN  CHARACTER := '7'
              CASE %8 THEN  CHARACTER := '8'
              CASE %9 THEN  CHARACTER := '9'
              CASE %A THEN  CHARACTER := 'A'
              CASE %B THEN  CHARACTER := 'B'
              CASE %C THEN  CHARACTER := 'C'
              CASE %D THEN  CHARACTER := 'D'
              CASE %E THEN  CHARACTER := 'E'
              CASE %F THEN  CHARACTER := 'F'
             FI
            FI
          END ASCII
```

```
VALUE PROCEDURE ( CHARACTER BYTE )
     RETURNS   ( MAGNITUDE BYTE )

! This routine returns the mathematical value of the ASCII character !
! passed to it.  MAGNITUDE will be returned with a value of zero for !
! undefined characters.                                              !

  ENTRY
    MAGNITUDE := %0
    IF CHARACTER
      CASE '0' THEN MAGNITUDE := %0
      CASE '1' THEN MAGNITUDE := %1
      CASE '2' THEN MAGNITUDE := %2
      CASE '3' THEN MAGNITUDE := %3
      CASE '4' THEN MAGNITUDE := %4
      CASE '5' THEN MAGNITUDE := %5
      CASE '6' THEN MAGNITUDE := %6
      CASE '7' THEN MAGNITUDE := %7
      CASE '8' THEN MAGNITUDE := %8
      CASE '9' THEN MAGNITUDE := %9
      CASE 'A' THEN MAGNITUDE := %A
      CASE 'B' THEN MAGNITUDE := %B
      CASE 'C' THEN MAGNITUDE := %C
      CASE 'D' THEN MAGNITUDE := %D
      CASE 'E' THEN MAGNITUDE := %E
      CASE 'F' THEN MAGNITUDE := %F
      FI
  END VALUE
```

```
            VALUE_LOOP PROCEDURE ( INPUT_STRING ASCII_PTR, MULTIPLER WORD )
                    RETURNS   ( MAGNITUDE WORD )

            ! This routine converts a string of ASCII characters into the numeric !
            ! value they represent.  The base of the representation is passed to  !
            ! the routine as MULTIPLER.  Thus any integer base from base 2 up can !
            ! be accomodated by this routine.  The routine assumes that the call- !
            ! ing routine has ensured that the input string contains only valid   !
            ! characters for the desired base.  If the number represented by the  !
            ! character string exceeds the PLZ word maximum value ( 65535.) the   !
            ! routine returns this maximum value as the MAGNITUDE represented by  !
            ! the input string.  This routine requires routine VALUE.             !

              LOCAL
                INDEX  BYTE
                FACTOR WORD

              ENTRY
  1             INDEX     := 0
  2             FACTOR    := 1
  3             MAGNITUDE := 0
                DO
  4               IF INPUT_STRING^[ INDEX ] = BLANK  ORIF  INDEX = 8  THEN  EXIT  FI
  5               MAGNITUDE := MAGNITUDE + ( FACTOR * WORD( VALUE( INPUT_STRING^[ INDEX ] ) ) )
  6               IF MAGNITUDE < FACTOR THEN     ! Overflow exists !
  7                 MAGNITUDE := 65535
  8                 EXIT
                  FI
  9               INDEX := INDEX + 1
 10               FACTOR := FACTOR * MULTIPLER
                OD
 11           END VALUE_LOOP
```

```
        PUTCH PROCEDURE( LOGICAL_UNIT CHARACTER  BYTE )

        ! This routine writes the character passed to it onto the specified !
        ! logical unit.  No action is taken for invalid operations.         !

          LOCAL
            LENGTH WORD
            RETURN_CODE BYTE
          ENTRY
1           LENGTH := 1
2           LENGTH, RETURN_CODE := PUTSEQ( LOGICAL_UNIT,*CHARACTER, LENGTH )
3         END PUTCH



         GETCH PROCEDURE ( LOGICAL_UNIT BYTE )
               RETURNS   ( CHARACTER    BYTE )

         ! This routine reads one character from the specified logical unit and  !
         ! returns that character to the calling routine.  If the reading is not !
         ! successful (return code <> 80H) the character is returned as a blank. !

         LOCAL
           RETURN_CODE BYTE
           LENGTH      WORD
         ENTRY
1          LENGTH, RETURN_CODE := GETSEQ( LOGICAL_UNIT, *CHARACTER, 1 )
2          IF RETURN_CODE <> OPERATION_OK THEN CHARACTER := BLANK FI
4        END GETCH
```

```
               GET_ASCII_CH PROCEDURE ( LOGICAL_UNIT BYTE )
                            RETURNS   ( CHARACTER    BYTE )

               ! This routine reads ( via GETCH ) one character from the specified    !
               ! logical_unit.  If CHARACTER is a printing or common cursor control   !
               ! character, CHARACTER is returned to the calling routine, otherwise a !
               ! blank ( 20H ) is returned.                                            !

                 ENTRY
                   DO
    1                CHARACTER := GETCH( LOGICAL_UNIT )
    2                IF ( CHARACTER >= %20  ANDIF  CHARACTER < %7F ) ORIF
                        CHARACTER = BELL      ORIF  CHARACTER = TAB  ORIF
                        CHARACTER = LINE_FEED ORIF  CHARACTER = CARRIAGE_RETURN ORIF
                        CHARACTER = ESCAPE
                      THEN EXIT FI
                   OD
    3            END GET_ASCII_CH
```

```
        PLACE_LOOP PROCEDURE( LOGICAL_UNIT BLANKING  BYTE, NUMBER INDEX DIVISOR  WORD )

        ! This routine outputs (via PUTCH) the character string which represents !
        ! the value of NUMBER.  The base of the representation is selected by    !
        ! the calling routine by specifying the value of DIVISOR.  The number of !
        ! output characters is determined by the value of INDEX.  INDEX begins   !
        ! with the value of the place of the most siginificant character.  The   !
        ! leading zeros of a representation are blanked if BLANKING is TRUE.     !

          LOCAL
            CHARACTER BYTE
            VALUE     WORD
          ENTRY
            DO
1             VALUE := NUMBER / INDEX
2             NUMBER := NUMBER MOD INDEX
3             BLANKING, CHARACTER := ASCII( VALUE, BLANKING )
4             PUTCH( LOGICAL_UNIT, CHARACTER )
5             IF INDEX = 1 THEN EXIT FI
6             INDEX := INDEX / DIVISOR
            OD
7         END PLACE_LOOP
```

```
          VALID_BINARY_CH PROCEDURE ( CHARACTER BYTE )
                          RETURNS   ( VALIDITY  BYTE )

          ! This routine checks to see whether the input character is a valid !
          ! character for expressing a binary number, ie a "0" or a "1".  If  !
          ! so VALIDITY is returned as TRUE, otherwise VALIDITY is FALSE.     !

            ENTRY
1             IF CHARACTER = '0' ORIF CHARACTER = '1'
2               THEN VALIDITY := TRUE
3               ELSE VALIDITY := FALSE
                FI
4           END VALID_BINARY_CH


          VALID_DECIMAL_CH PROCEDURE ( CHARACTER BYTE )
                           RETURNS   ( VALIDITY  BYTE )

          ! This routine checks to see whether the input character is a valid !
          ! character for expressing a decimal number, ie a "0" to a "9".  If !
          ! so VALIDITY is returned as TRUE, otherwise VALIDITY is FALSE.     !

            ENTRY
1             IF CHARACTER = '0' ORIF CHARACTER = '1' ORIF
                 CHARACTER = '2' ORIF CHARACTER = '3' ORIF
                 CHARACTER = '4' ORIF CHARACTER = '5' ORIF
                 CHARACTER = '6' ORIF CHARACTER = '7' ORIF
                 CHARACTER = '8' ORIF CHARACTER = '9'
2               THEN VALIDITY := TRUE
3               ELSE VALIDITY := FALSE
                FI
4           END VALID_DECIMAL_CH
```

```
       VALID_HEX_CH PROCEDURE ( CHARACTER BYTE )
                    RETURNS   ( VALIDITY  BYTE )

       ! This routine checks to see whether the input character is a valid !
       ! character for expressing a hexadecimal number, ie a "0" to a "F". !
       ! If so VALIDITY is returned as TRUE, otherwise VALIDITY is FALSE.  !

         ENTRY
1          IF CHARACTER = '0' ORIF CHARACTER = '1' ORIF
              CHARACTER = '2' ORIF CHARACTER = '3' ORIF
              CHARACTER = '4' ORIF CHARACTER = '5' ORIF
              CHARACTER = '6' ORIF CHARACTER = '7' ORIF
              CHARACTER = '8' ORIF CHARACTER = '9' ORIF
              CHARACTER = 'A' ORIF CHARACTER = 'B' ORIF
              CHARACTER = 'C' ORIF CHARACTER = 'D' ORIF
              CHARACTER = 'E' ORIF CHARACTER = 'F'
2            THEN VALIDITY := TRUE
3            ELSE VALIDITY := FALSE
             FI
4        END VALID_HEX_CH
```

```
        GLOBAL   !!!!!!!!!! routines callable by routines in other modules !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

           !!!!!!!!!!! beginning of write statements !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!



           WRITE PROCEDURE ( LOGICAL_UNIT BYTE, TEXT_POINTER PBYTE )

           ! This routine outputs to the specified logical unit the string of ASCII !
           ! characters pointed to by TEXT_POINTER.  No carriage return is output.  !

             LOCAL
               LENGTH WORD
               RETURN_CODE BYTE
               PINDEX PBYTE
             ENTRY
1              LENGTH := 0
2              PINDEX := TEXT_POINTER
               DO
3                IF PINDEX^ = CARRIAGE_RETURN THEN EXIT FI
4                LENGTH := LENGTH + 1
5                PINDEX := INC PINDEX
               OD
6              IF LENGTH > 0 THEN LENGTH, RETURN_CODE := PUTSEQ( LOGICAL_UNIT, TEXT_POINTER, LENGTH ) FI
8            END WRITE


           WRITELN PROCEDURE ( LOGICAL_UNIT BYTE, TEXT_POINTER PBYTE )

           ! This routine outputs to the specified logical unit the string of ASCII !
           ! characters pointed to by TEXT_POINTER.   A carriage return is output.  !

             ENTRY
1              WRITE( LOGICAL_UNIT, TEXT_POINTER )
2              PUTCH( LOGICAL_UNIT, '%R' )
3            END WRITELN
```

```
        WRITE_DBYTE PROCEDURE ( LOGICAL_UNIT NUMBER BYTE )

        ! This routine outputs the ASCII characters representing the value of a !
        ! variable of type BYTE in the format ddd. where d=(0,1,..9," ").  No   !
        ! carriage return is output.  Leading zeros are blanked.                !

          LOCAL
            BLANKING BYTE
            INDEX  WORD

          ENTRY
1           BLANKING := TRUE
2           INDEX := 100
3           PLACE_LOOP( LOGICAL_UNIT, BLANKING, WORD( NUMBER ), INDEX, 10 )
4           PUTCH( LOGICAL_UNIT, '.' )
5         END WRITE_DBYTE


        WRITELN_DBYTE PROCEDURE ( LOGICAL_UNIT NUMBER BYTE )

        ! This routine outputs the ASCII characters representing the value of a !
        ! variable of type BYTE in the format ddd. where d=(0,1,..9," ").  A    !
        ! carriage return is output.  Leading zeros are blanked.                !

          ENTRY
1           WRITE_DBYTE( LOGICAL_UNIT, NUMBER )
2           PUTCH( LOGICAL_UNIT, '%R' )
3         END WRITELN_DBYTE
```

```
        WRITE_HBYTE PROCEDURE ( LOGICAL_UNIT NUMBER BYTE )

        ! This routine outputs the ASCII characters representing the value of a !
        ! variable of type BYTE in the format hh. where h=(0,1,..F," ").  No    !
        ! carriage return is output.  Leading zeros are not blanked.            !

          LOCAL
            BLANKING  BYTE
            INDEX     WORD

          ENTRY
1           BLANKING := FALSE
2           INDEX := %10
3           PLACE_LOOP( LOGICAL_UNIT, BLANKING, WORD( NUMBER ), INDEX, %10 )
4           PUTCH( LOGICAL_UNIT, 'H' )
5         END WRITE_HBYTE


        WRITELN_HBYTE PROCEDURE ( LOGICAL_UNIT NUMBER BYTE )

        ! This routine outputs the ASCII characters representing the value of a !
        ! variable of type BYTE in the format hh. where h=(0,1,..F," ").  A     !
        ! carriage return is output.  Leading zeros are not blanked.            !

          ENTRY
1           WRITE_HBYTE( LOGICAL_UNIT, NUMBER )
2           PUTCH( LOGICAL_UNIT, '%R' )
3         END WRITELN_HBYTE
```

```
          WRITE_BBYTE PROCEDURE ( LOGICAL_UNIT NUMBER BYTE )

          ! This routine outputs the ASCII characters representing the value of a !
          ! variable of type BYTE in the format bbbbbbbbB. where b=(0,1).  No     !
          ! carriage return is output.  Leading zeros are not blanked.            !

            LOCAL
              BLANKING  BYTE
              INDEX     WORD
            ENTRY
1             BLANKING := FALSE
2             INDEX := %80
3             PLACE_LOOP( LOGICAL_UNIT, BLANKING, WORD( NUMBER ), INDEX, %2 )
4             PUTCH( LOGICAL_UNIT, 'B' )
5           END WRITE_BBYTE


          WRITELN_BBYTE PROCEDURE ( LOGICAL_UNIT NUMBER BYTE )

          ! This routine outputs the ASCII characters representing the value of a !
          ! variable of type BYTE in the format bbbbbbbbB. where b=(0,1).  A      !
          ! carriage return is output.  Leading zeros are not blanked.            !

            ENTRY
1             WRITE_DBYTE( LOGICAL_UNIT, NUMBER )
2             PUTCH( LOGICAL_UNIT, '%R' )
3           END WRITELN_BBYTE
```

```
            WRITE_IBYTE PROCEDURE ( LOGICAL_UNIT FLAG  BYTE )

            ! This routine outputs the logical value of a variable of type BYTE to    !
            ! the specified logical unit.  The output is one of the following:        !
            ! " TRUE", "FALSE", " UNDF".  No carriage return is output.               !

              ENTRY
1               IF FLAG = TRUE
2                 THEN   WRITE( LOGICAL_UNIT, #'TRUE %R' )
3                 ELSE IF FLAG = FALSE
4                   THEN WRITE( LOGICAL_UNIT, #'FALSE%R' )
5                   ELSE WRITE( LOGICAL_UNIT, #'UNDF %R' )
                    FI
                  FI
6             END WRITE_IBYTE


            WRITELN_IBYTE PROCEDURE ( LOGICAL_UNIT FLAG  BYTE )

            ! This routine outputs the logical value of a variable of type BYTE to    !
            ! the specified logical unit.  The output is one of the following:        !
            ! " TRUE", "FALSE", " UNDF".   A carriage return is output.               !

              ENTRY
1               WRITE_IBYTE( LOGICAL_UNIT, FLAG )
2               PUTCH( LOGICAL_UNIT, '%R' )
3             END WRITELN_IBYTE
```

```
            WRITE_DINTEGER PROCEDURE ( LOGICAL_UNIT BYTE, IN_INTEGER INTEGER )

            ! This routine outputs the ASCII characters representing the value of a !
            ! variable of type INTEGER in the format sdddddd. where d=(0,1,..9," "), !
            ! and s=(" ","-").  No carriage return is output.  Leading zeros are    !
            ! blanked.                                                              !

              LOCAL
                INDEX, NUMBER  WORD
                BLANKING       BYTE

              ENTRY
 1              BLANKING := TRUE
 2              INDEX := 10000
 3              IF IN_INTEGER < 0
                  THEN
 4                  PUTCH( LOGICAL_UNIT, '-' )
 5                  IF IN_INTEGER = -32768
 6                    THEN NUMBER := 32768
 7                    ELSE NUMBER := WORD( (-1) * IN_INTEGER )
                      FI
 8                ELSE PUTCH( LOGICAL_UNIT, ' ' )
                  FI
 9              PLACE_LOOP( LOGICAL_UNIT, BLANKING, NUMBER, INDEX, 10 )
10              PUTCH( LOGICAL_UNIT, '.' )
11            END WRITE_DINTEGER


            WRITELN_DINTEGER PROCEDURE ( LOGICAL_UNIT BYTE, IN_INTEGER INTEGER )

            ! This routine outputs the ASCII characters representing the value of a !
            ! variable of type INTEGER in the format sdddddd. where d=(0,1,..9," "), !
            ! and s=(" ","-").   A carriage return is output.  Leading zeros are    !
            ! blanked.                                                              !

              ENTRY
 1              WRITE_DINTEGER( LOGICAL_UNIT, IN_INTEGER )
 2              PUTCH( LOGICAL_UNIT, '%R' )
 3            END WRITELN_DINTEGER
```

```
WRITE_DWORD PROCEDURE ( LOGICAL_UNIT BYTE, NUMBER WORD )

! This routine outputs the ASCII characters representing the value of a !
! variable of type WORD in the format ddddd. where d=(0,1,..9," ").      !
! No carriage return is output.    Leading zeros are blanked.           !

  LOCAL
    BLANKING  BYTE
    INDEX     WORD

  ENTRY
1   BLANKING := TRUE
2   INDEX := 10000
3   PLACE_LOOP( LOGICAL_UNIT, BLANKING, NUMBER, INDEX, 10 )
4   PUTCH( LOGICAL_UNIT, '.' )
5 END WRITE_DWORD


WRITELN_DWORD PROCEDURE ( LOGICAL_UNIT BYTE, NUMBER WORD )

! This routine outputs the ASCII characters representing the value of a !
! variable of type WORD in the format ddddd. where d=(0,1,..9," ").      !
! A carriage return is output.    Leading zeros are blanked.            !

  ENTRY
1   WRITE_DWORD( LOGICAL_UNIT, NUMBER )
2   PUTCH( LOGICAL_UNIT, '%R' )
3 END WRITELN_DWORD
```

```
WRITE_HWORD PROCEDURE ( LOGICAL_UNIT BYTE, NUMBER WORD )

! This routine outputs the ASCII characters representing the value of a !
! variable of type WORD in the format hhhhH where h=F(0,1,..9," ").     !
! No carriage return is output.    Leading zeros are not blanked.       !

  LOCAL
    BLANKING  BYTE
    INDEX     WORD

  ENTRY
    BLANKING := FALSE
    INDEX := %1000
    PLACE_LOOP( LOGICAL_UNIT, BLANKING, NUMBER, INDEX, %10 )
    PUTCH( LOGICAL_UNIT, 'H' )
  END WRITE_HWORD


WRITELN_HWORD PROCEDURE ( LOGICAL_UNIT BYTE, NUMBER WORD )

! This routine outputs the ASCII characters representing the value of a !
! variable of type WORD in the format hhhhH where h=F(0,1,..9," ").     !
! A carriage return is output.     Leading zeros are not blanked.       !

  ENTRY
    WRITE_DWORD( LOGICAL_UNIT, NUMBER )
    PUTCH( LOGICAL_UNIT, '%R' )
  END WRITELN_HWORD
```

```
        WRITE_POINTER PROCEDURE ( LOGICAL_UNIT BYTE, LOCATION WORD )

        ! This routine outputs to the specified logical unit the ASCII characters  !
        ! which represent the memory location pointed to by LOCATION.  The output  !
        ! is in the format hhhh^ where h=(0,1,...F). No carriage return is output. !

          LOCAL
            BLANKING  BYTE
            INDEX     WORD

          ENTRY
1           PUTCH( LOGICAL_UNIT, '^' )
2           BLANKING := FALSE
3           INDEX := %1000
4           PLACE_LOOP( LOGICAL_UNIT, BLANKING, LOCATION, INDEX, %10 )
5         END WRITE_POINTER


        WRITELN_POINTER PROCEDURE ( LOGICAL_UNIT BYTE, LOCATION WORD )

        ! This routine outputs to the specified logical unit the ASCII characters  !
        ! which represent the memory location pointed to by LOCATION.  The output  !
        ! is in the format hhhh^ where h=(0,1,...F).  A carriage return is output. !

          ENTRY
1           WRITE_POINTER( LOGICAL_UNIT, LOCATION )
2           PUTCH( LOGICAL_UNIT, '%R' )
3         END WRITELN_POINTER
```

```
WRITE_RCODE PROCEDURE( LOGICAL_UNIT RETURN_CODE BYTE )

! This PLZ debugging routine outputs the standard RIO return code defini- !
! tion as a message to the logical unit specified by the calling routine. !
! A carriage return is not output at the end of the message.              !

  ENTRY
    IF RETURN_CODE
      CASE %40 THEN WRITE( LOGICAL_UNIT, #'Invalid Drive Name %R' )
      CASE %41 THEN WRITE( LOGICAL_UNIT, #'Invalid or Inactive Device %R' )
      CASE %42 THEN WRITE( LOGICAL_UNIT, #'Invalid Unit %R' )
      CASE %43 THEN WRITE( LOGICAL_UNIT, #'Memory Protect Violation %R' )
      CASE %44 THEN WRITE( LOGICAL_UNIT, #'Missing or Invalid Operand(s) %R' )
      CASE %45 THEN WRITE( LOGICAL_UNIT, #'System Error %R' )
      CASE %46 THEN WRITE( LOGICAL_UNIT, #'Illegal File Name %R' )
      CASE %47 THEN WRITE( LOGICAL_UNIT, #'Non-existant Command %R' )
      CASE %48 THEN WRITE( LOGICAL_UNIT, #'Illegal File Type %R' )
      CASE %49 THEN WRITE( LOGICAL_UNIT, #'Program Abort %R' )
      CASE %4A THEN WRITE( LOGICAL_UNIT, #'Insufficient Memory %R' )
      CASE %4B THEN WRITE( LOGICAL_UNIT, #'Missing or Invalid File Properities %R' )
      CASE %4C THEN WRITE( LOGICAL_UNIT, #'I/O Error %R' )
      CASE %80 THEN WRITE( LOGICAL_UNIT, #'Operation Complete %R' )
      CASE %81 THEN WRITE( LOGICAL_UNIT, #'Directory Format Error %R' )
      CASE %82 THEN WRITE( LOGICAL_UNIT, #'Scratch File Created %R' )
      CASE %83 THEN WRITE( LOGICAL_UNIT, #'File Name Truncated %R' )
      CASE %84 THEN WRITE( LOGICAL_UNIT, #'Attribute List Truncated %R' )
      CASE %C1 THEN WRITE( LOGICAL_UNIT, #'Invalid Operation (request) %R' )
      CASE %C2 THEN WRITE( LOGICAL_UNIT, #'Device Not Ready %R' )
      CASE %C3 THEN WRITE( LOGICAL_UNIT, #'Write Protection %R' )
      CASE %C4 THEN WRITE( LOGICAL_UNIT, #'Sector Address Error %R' )
      CASE %C5 THEN WRITE( LOGICAL_UNIT, #'Seek Error %R' )
      CASE %C6 THEN WRITE( LOGICAL_UNIT, #'Data Transfer Error %R' )
      CASE %C7 THEN WRITE( LOGICAL_UNIT, #'File Not Found %R' )
      CASE %C9 THEN WRITE( LOGICAL_UNIT, #'End of File Error %R' )
```

```
            CASE %CA THEN WRITE( LOGICAL_UNIT, #'Pointer Check Error %R' )
            CASE %CB THEN WRITE( LOGICAL_UNIT, #'File Not Open %R' )
            CASE %CC THEN WRITE( LOGICAL_UNIT, #'Unit Already Active (open) %R' )
            CASE %CD THEN WRITE( LOGICAL_UNIT, #'Assign Buffer Full %R' )
            CASE %CE THEN WRITE( LOGICAL_UNIT, #'Invalid Drive Specification %R' )
            CASE %CF THEN WRITE( LOGICAL_UNIT, #'Logical Unit Table Full %R' )

            CASE %D0 THEN WRITE( LOGICAL_UNIT, #'Duplicate File %R' )
            CASE %D1 THEN WRITE( LOGICAL_UNIT, #'Diskette ID Error %R' )
            CASE %D2 THEN WRITE( LOGICAL_UNIT, #'Invalid Attributes %R' )
            CASE %D3 THEN WRITE( LOGICAL_UNIT, #'Disk Is Full %R' )
            CASE %D4 THEN WRITE( LOGICAL_UNIT, #'File Not Found in Proper Directory Record %R' )
            CASE %D5 THEN WRITE( LOGICAL_UNIT, #'Beginning of File Error %R' )
            CASE %D6 THEN WRITE( LOGICAL_UNIT, #'File Already Open on Other Unit %R' )
            CASE %D7 THEN WRITE( LOGICAL_UNIT, #'Invalid Rename to Scratch File %R' )
            CASE %D8 THEN WRITE( LOGICAL_UNIT, #'File Locked %R' )
            CASE %D9 THEN WRITE( LOGICAL_UNIT, #'Invalid Open Request %R' )
            CASE %DA THEN WRITE( LOGICAL_UNIT, #'Insufficient Memory for Allocation Maps %R' )
            FI
        END WRITE_RCODE


      WRITELN_RCODE PROCEDURE ( LOGICAL_UNIT RETURN_CODE BYTE )

    ! This PLZ debugging routine outputs the standard RIO return code defini- !
    ! tion as a message to the logical unit specified by the calling routine. !
    ! A carriage return is output at the end of the message.                  !

        ENTRY
          WRITE_RCODE( LOGICAL_UNIT RETURN_CODE )
          WRITELN( LOGICAL_UNIT, #'%R' )
        END WRITELN_RCODE
```

```
            !||||||||||| beginning of read statements ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||


            READLN PROCEDURE ( LOGICAL_UNIT BYTE, TEXT_POINTER PBYTE )
                   RETURNS   ( OUT_POINTER PBYTE )

            ! This routine reads in a string of text characters until and including !
            ! the first carriage return.  A pointer to the beginning of this string !
            ! TEXT_POINTER is passed to the routine; the routine passes back a      !
            ! pointer to the end of the string, OUT_POINTER.  The calling routine   !
            ! must have dimensioned its text buffer large enough to accomodate the  !
            ! input string.                                                         !

              LOCAL
                PINDEX PBYTE
              ENTRY
 1              PINDEX := TEXT_POINTER
                DO
 2                PINDEX^ := GET_ASCII_CH( LOGICAL_UNIT )
 3                IF PINDEX^ = CARRIAGE_RETURN THEN EXIT FI
 4                PINDEX := INC PINDEX
                OD
 5              OUT_POINTER := PINDEX
 6            END READLN
```

```
          READ_HBYTE PROCEDURE ( LOGICAL_UNIT BYTE )
                     RETURNS   ( NUMBER       BYTE )

          ! This routine reads in from the specified logical unit one or two ASCII !
          ! characters in the format "hh", where h=("0","1",... "F").  NUMBER is   !
          ! returned to the calling routine with the hexadecimal value represented !
          ! by the input characters.                                               !

            LOCAL
              FIRST_TERM SECOND_TERM BYTE
            ENTRY
              DO
1               FIRST_TERM := GET_ASCII_CH( LOGICAL_UNIT )
2               IF VALID_HEX_CH( FIRST_TERM) = TRUE THEN EXIT FI
              OD
3             SECOND_TERM := GET_ASCII_CH( LOGICAL_UNIT )
4             IF VALID_HEX_CH( SECOND_TERM ) <> TRUE
5               THEN NUMBER := VALUE( FIRST_TERM )
6               ELSE NUMBER := ( %10 * VALUE( FIRST_TERM ) ) + VALUE( SECOND_TERM )
                FI
7           END READ_HBYTE
```

```
        READ_DBYTE PROCEDURE ( LOGICAL_UNIT BYTE )
                   RETURNS   ( NUMBER       BYTE )

        ! This routine reads in from the specified logical unit 1 to 3 ASCII     !
        ! characters in the format "ddd", where d=("0","1",... "9").  NUMBER is  !
        ! returned to the calling routine with the decimal value represented by  !
        ! the input characters.                                                  !

          LOCAL
            FIRST_TERM SECOND_TERM THIRD_TERM BYTE
          ENTRY
            DO
 1            FIRST_TERM := GET_ASCII_CH( LOGICAL_UNIT )
 2            IF VALID_DECIMAL_CH( FIRST_TERM) = TRUE THEN EXIT FI
            OD
 3          SECOND_TERM := GET_ASCII_CH( LOGICAL_UNIT )
 4          IF VALID_DECIMAL_CH( SECOND_TERM ) <> TRUE
 5            THEN NUMBER := VALUE( FIRST_TERM )
              ELSE
 6              THIRD_TERM := GET_ASCII_CH( LOGICAL_UNIT )
 7              IF VALID_DECIMAL_CH( THIRD_TERM ) <> TRUE
 8                THEN NUMBER := ( 10 * VALUE( FIRST_TERM ) ) + VALUE( SECOND_TERM )
 9                ELSE NUMBER := ( 100 * VALUE( FIRST_TERM ) ) + ( 10 * VALUE( SECOND_TERM ) ) + VALUE( THIRD_TERM )
                FI
              FI
10        END READ_DBYTE
```

```
         READ_BBYTE PROCEDURE ( LOGICAL_UNIT BYTE )
                   RETURNS   ( NUMBER BYTE )

         ! This routine reads in 1 to 8 characters from the specified logical !
         ! unit in the format bbbbbbbb to determine a value for a variable of !
         ! type BYTE where b=( 0, 1 ) and there is at least one b.            !

           LOCAL
             INPUT_STRING ASCII_STR
             INDEX CHARACTER BYTE
           ENTRY
             DO
 1             CHARACTER := GET_ASCII_CH( LOGICAL_UNIT )
 2             IF CHARACTER = '0' ORIF CHARACTER = '1' THEN EXIT FI
             OD
 3           INPUT_STRING[ 0 ] := CHARACTER
 4           INDEX := 0
             DO
 5             CHARACTER := GET_ASCII_CH( LOGICAL_UNIT )
 6             IF CHARACTER <> '0' ANDIF CHARACTER <> '1' THEN
 7               INPUT_STRING[ INDEX ] := BLANK
 8               EXIT
                 FI
 9             INPUT_STRING[ INDEX ] := CHARACTER
10             INDEX := INDEX + 1
11             IF INDEX = 8 THEN EXIT FI
             OD
12           NUMBER := BYTE( VALUE_LOOP(  INPUT_STRING, 2 ) )
13         END READ_BBYTE
```

```



       READ_IBYTE PROCEDURE ( LOGICAL_UNIT BYTE )
                  RETURNS   ( TRUTH        BYTE )

       ! This procedure reads a character from the specified logical unit.   !
       ! Depending upon the character read, TRUTH is returned as either TRUE !
       ! or FALSE.                                                            !

       LOCAL
         CHARACTER BYTE
       ENTRY
         DO
1          CHARACTER := GET_ASCII_CH( LOGICAL_UNIT )
2          IF CHARACTER
             CASE 'T','t','1' THEN
3              TRUTH := TRUE
4              EXIT
             CASE 'F','f','0' THEN
5              TRUTH := FALSE
6              EXIT
             FI
         OD
7      END READ_IBYTE
```

```
          READ_DINTEGER PROCEDURE ( LOGICAL_UNIT BYTE )
                        RETURNS   ( NUMBER INTEGER )

          ! This routine reads in a value for a INTEGER in the format sddddd.  !
          ! where s = ( ' ','+','-' ), d = ( 0,1,... 9 ), and there is at least !
          ! one d.                                                              !

            LOCAL
              INPUT_STRING ASCII_STR
              INDEX CHARACTER SIGN BYTE

            ENTRY
              DO
 1              SIGN := GET_ASCII_CH( LOGICAL_UNIT )
 2              IF SIGN = ' ' ORIF SIGN = '+' ORIF SIGN = '-' THEN
 3                CHARACTER := GET_ASCII_CH( LOGICAL_UNIT )
 4                IF VALID_DECIMAL_CH( CHARACTER ) = TRUE THEN EXIT FI
                  FI
              OD
 5            INPUT_STRING[ 0 ] := CHARACTER
 6            INDEX := 0
              DO
 7              CHARACTER := GET_ASCII_CH( LOGICAL_UNIT )
 8              IF VALID_DECIMAL_CH( CHARACTER ) = FALSE THEN
 9                INPUT_STRING[ INDEX ] := BLANK
10                EXIT
                  FI
11              INPUT_STRING[ INDEX ] := CHARACTER
12              INDEX := INDEX + 1
13              IF INDEX = 5 THEN
14                INPUT_STRING[ INDEX ] := BLANK
15                EXIT
                  FI
              OD
```

```
    NUMBER := INTEGER( VALUE_LOOP(  INPUT_STRING, 10 ) )
    IF NUMBER < 32768          ! Does the number overflow the integer range ? !
      THEN                     !   No, the number does not overflow integer range. !
        IF SIGN = '-' THEN NUMBER := NUMBER * (-1) FI  ! If the number is negative correct the sign. !
      ELSE IF SIGN = '-'       !   Yes, number overflows.  Is the number negative ?  !
        THEN NUMBER := -32768 !     Yes, output the maximum negative integer value. !
        ELSE NUMBER :=  32767 !     No, output the mximum positive integer value.    !
        FI
      FI
  END READ_DINTEGER
```

```
             READ_HWORD PROCEDURE ( LOGICAL_UNIT BYTE )
                     RETURNS   ( NUMBER WORD )

             ! This routine reads in a value for a WORD in the format hhhhH where !
             ! h = ( 0,1,... F ) and there is at least one h.                      !

               LOCAL
                 INPUT_STRING ASCII_STR
                 INDEX CHARACTER BYTE

               ENTRY
                 DO
 1                 CHARACTER := GET_ASCII_CH( LOGICAL_UNIT )
 2                 IF VALID_HEX_CH( CHARACTER ) = TRUE THEN EXIT FI
                 OD
 3               INPUT_STRING[ 0 ] := CHARACTER
 4               INDEX := 0
                 DO
 5                 CHARACTER := GET_ASCII_CH( LOGICAL_UNIT )
 6                 IF VALID_HEX_CH( CHARACTER ) = FALSE THEN
 7                   INPUT_STRING[ INDEX ] := BLANK
 8                   EXIT
                     FI
 9                 INPUT_STRING[ INDEX ] := CHARACTER
10                 INDEX := INDEX + 1
11                 IF INDEX = 4 THEN
12                   INPUT_STRING[ INDEX ] := BLANK
13                   EXIT
                     FI
                 OD
13               NUMBER := VALUE_LOOP(  INPUT_STRING, %10 )
14             END READ_HWORD
```

```
          READ_DWORD PROCEDURE ( LOGICAL_UNIT BYTE )
                     RETURNS    ( NUMBER WORD )

          ! This routine reads in a value for a WORD in the format dddddd.  !
          ! where d = ( 0,1,... 9 ) and there is at least one d.            !

            LOCAL
              INPUT_STRING ASCII_STR
              INDEX CHARACTER BYTE

            ENTRY
              DO
1               CHARACTER := GET_ASCII_CH( LOGICAL_UNIT )
2               IF VALID_DECIMAL_CH( CHARACTER ) = TRUE THEN EXIT FI
              OD
3             INPUT_STRING[ 0 ] := CHARACTER
4             INDEX := 0
              DO
5               CHARACTER := GET_ASCII_CH( LOGICAL_UNIT )
6               IF VALID_DECIMAL_CH( CHARACTER ) = FALSE THEN
7                 INPUT_STRING[ INDEX ] := BLANK
8                 EXIT
                  FI
9               INPUT_STRING[ INDEX ] := CHARACTER
10              INDEX := INDEX + 1
11              IF INDEX = 6 THEN
12                INPUT_STRING[ INDEX ] := BLANK
13                EXIT
                  FI
              OD
13            NUMBER := VALUE_LOOP(  INPUT_STRING, 10 )
14          END READ_DWORD



        END ENHANCEMENTS
```

```
END OF COMPILATION:    0 ERROR(S)      0 WARNING(S)
  980 DATA BYTES   2119 Z-CODE BYTES    SYMBOL TABLE 18% FULL
```

```
PLZSYS 3.0  810107.1249
        DEBUGS MODULE

            ! This module contains routines for debugging PLZ programs.  By linking   !
            ! module DEBUGS in with the program under test and declairing the desired !
            ! routines EXTERNAL these debug routines may be called for diagonistics.  !
            ! Note that these routines output their messages to the system console    !
            ! output device and that the ENHANCEMENTS module ( which contains proce-  !
            ! dures WRITE and WRITELN ) must also be linked.                          !

          CONSTANT
            CONSOLE_OUT := 2
            ! "%R" represents an ASCII carriage return. !

          TYPE
            PBYTE ^BYTE

          EXTERNAL

            WRITE   PROCEDURE ( LOGICAL_UNIT BYTE, TEXT_POINTER PBYTE )
            WRITELN PROCEDURE ( LOGICAL_UNIT BYTE, TEXT_POINTER PBYTE )


        !!!!!!!!!!! beginning of globals !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!


        GLOBAL


          WRITE_RCODE PROCEDURE( RETURN_CODE BYTE )

            ! This PLZ debugging routine outputs the standard RIO return code defini- !
            ! tion as a message to the system console output device (CONOUT).  A car- !
            ! riage return is not output at the end of the message.                   !

          ENTRY
            IF RETURN_CODE
              CASE %40 THEN WRITE( CONSOLE_OUT, #'Invalid Drive Name %R' )
              CASE %41 THEN WRITE( CONSOLE_OUT, #'Invalid or Inactive Device %R' )
              CASE %42 THEN WRITE( CONSOLE_OUT, #'Invalid Unit %R' )
              CASE %43 THEN WRITE( CONSOLE_OUT, #'Memory Protect Violation %R' )
              CASE %44 THEN WRITE( CONSOLE_OUT, #'Missing or Invalid Operand(s) %R' )
              CASE %45 THEN WRITE( CONSOLE_OUT, #'System Error %R' )
              CASE %46 THEN WRITE( CONSOLE_OUT, #'Illegal File Name %R' )
              CASE %47 THEN WRITE( CONSOLE_OUT, #'Non-existant Command %R' )
              CASE %48 THEN WRITE( CONSOLE_OUT, #'Illegal File Type %R' )
              CASE %49 THEN WRITE( CONSOLE_OUT, #'Program Abort %R' )
              CASE %4A THEN WRITE( CONSOLE_OUT, #'Insufficient Memory %R' )
              CASE %4B THEN WRITE( CONSOLE_OUT, #'Missing or Invalid File Properities %R' )
              CASE %4C THEN WRITE( CONSOLE_OUT, #'I/O Error %R' )
              CASE %80 THEN WRITE( CONSOLE_OUT, #'Operation Complete %R' )
              CASE %81 THEN WRITE( CONSOLE_OUT, #'Directory Format Error %R' )
              CASE %82 THEN WRITE( CONSOLE_OUT, #'Scratch File Created %R' )
```

```
          CASE %83 THEN WRITE( CONSOLE_OUT, #'File Name Truncated %R' )
          CASE %84 THEN WRITE( CONSOLE_OUT, #'Attribute List Truncated %R' )
          CASE %C1 THEN WRITE( CONSOLE_OUT, #'Invalid Operation (request) %R' )
          CASE %C2 THEN WRITE( CONSOLE_OUT, #'Device Not Ready %R' )
          CASE %C3 THEN WRITE( CONSOLE_OUT, #'Write Protection %R' )
          CASE %C4 THEN WRITE( CONSOLE_OUT, #'Sector Address Error %R' )
          CASE %C5 THEN WRITE( CONSOLE_OUT, #'Seek Error %R' )
          CASE %C6 THEN WRITE( CONSOLE_OUT, #'Data Transfer Error %R' )
          CASE %C7 THEN WRITE( CONSOLE_OUT, #'File Not Found %R' )
          CASE %C9 THEN WRITE( CONSOLE_OUT, #'End of File Error %R' )
          CASE %CA THEN WRITE( CONSOLE_OUT, #'Pointer Check Error %R' )
          CASE %CB THEN WRITE( CONSOLE_OUT, #'File Not Open %R' )
          CASE %CC THEN WRITE( CONSOLE_OUT, #'Unit Already Active (open) %R' )
          CASE %CD THEN WRITE( CONSOLE_OUT, #'Assign Buffer Full %R' )
          CASE %CE THEN WRITE( CONSOLE_OUT, #'Invalid Drive Specification %R' )
          CASE %CF THEN WRITE( CONSOLE_OUT, #'Logical Unit Table Full %R' )
          CASE %D0 THEN WRITE( CONSOLE_OUT, #'Duplicate File %R' )
          CASE %D1 THEN WRITE( CONSOLE_OUT, #'Diskette ID Error %R' )
          CASE %D2 THEN WRITE( CONSOLE_OUT, #'Invalid Attributes %R' )
          CASE %D3 THEN WRITE( CONSOLE_OUT, #'Disk Is Full %R' )
          CASE %D4 THEN WRITE( CONSOLE_OUT, #'File Not Found in Proper Directory Record %R' )
          CASE %D5 THEN WRITE( CONSOLE_OUT, #'Beginning of File Error %R' )
          CASE %D6 THEN WRITE( CONSOLE_OUT, #'File Already Open on Other Unit %R' )
          CASE %D7 THEN WRITE( CONSOLE_OUT, #'Invalid Rename to Scratch File %R' )
          CASE %D8 THEN WRITE( CONSOLE_OUT, #'File Locked %R' )
          CASE %D9 THEN WRITE( CONSOLE_OUT, #'Invalid Open Request %R' )
          CASE %DA THEN WRITE( CONSOLE_OUT, #'Insufficient Memory for Allocation Maps %R' )
          OT
      END WRITE_RCODE


      WRITELN_RCODE PROCEDURE ( RETURN_CODE BYTE )

        ! This PLZ debugging routine outputs the standard RIO return code defini- !
        ! tion as a message to the system console output device (CONOUT).  A car- !
        ! riage return is output at the end of the message.                        !

        ENTRY
          WRITE_RCODE( RETURN_CODE )
          WRITELN( CONSOLE_OUT, #'%R' )
        END WRITELN_RCODE



    END DEBUGS


END OF COMPILATION:    0 ERROR(S)      0 WARNING(S)
  962 DATA BYTES    491 Z-CODE BYTES    SYMBOL TABLE  23 FULL
```

```
PLZSYS 3.0  810210.1650
TEST_IT MODULE                                      ! 10 February 1981 !

    ! This routine is solely for the testing of the input output routines !
    ! in the ENHANCEMENTS module.  This routine begins with a series of    !
    ! write and writelns.  Next the read statements are tried, using the   !
    ! write and writeln statements to verify receipt of the data via the   !
    ! the read statements.  All types of read and write statements are     !
    ! tested by this routine.                                              !

  CONSTANT    !|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||!

    CONIN  := 1      ! Logical unit number for the console input. !
    CONOUT := 2      ! Logical unit number for the console output. !
    SYSLST := 3      ! Logical unit number for the system list device. !

    CARRIAGE_RETURN := %0D ! ASCII numberic for carriage return. PLZ %R !

  TYPE        !|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||!

    PBYTE ^BYTE
    ASCII_STR ARRAY[ 8 BYTE ]
    ASCII_PTR ^ASCII_STR


  EXTERNAL    !|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||!

    WRITE PROCEDURE ( LOGICAL_UNIT BYTE, POINTER PBYTE )

    WRITELN PROCEDURE ( LOGICAL_UNIT BYTE, POINTER PBYTE )

    WRITE_DBYTE PROCEDURE ( LOGICAL_UNIT BYTE, NUMBER BYTE )

    WRITELN_DBYTE PROCEDURE ( LOGICAL_UNIT BYTE, NUMBER BYTE )

    WRITE_HBYTE PROCEDURE ( LOGICAL_UNIT BYTE, NUMBER BYTE )

    WRITELN_HBYTE PROCEDURE ( LOGICAL_UNIT BYTE, NUMBER BYTE )

    WRITE_BBYTE PROCEDURE ( LOGICAL_UNIT BYTE, NUMBER BYTE )

    WRITELN_BBYTE PROCEDURE ( LOGICAL_UNIT BYTE, NUMBER BYTE )

    WRITE_LBYTE PROCEDURE( LOGICAL_UNIT BYTE, NUMBER BYTE )

    WRITELN_LBYTE PROCEDURE( LOGICAL_UNIT BYTE, NUMBER BYTE )

    ...............
```

```
!!!!!!!!!!!!!!!!

WRITE_DINTEGER PROCEDURE( LOGICAL_UNIT BYTE, NUMBER INTEGER )

WRITELN_DINTEGER PROCEDURE( LOGICAL_UNIT BYTE, NUMBER INTEGER )

!!!!!!!!!!!!

WRITE_DWORD PROCEDURE( LOGICAL_UNIT BYTE, NUMBER WORD )

WRITELN_DWORD PROCEDURE( LOGICAL_UNIT BYTE, NUMBER WORD )

WRITE_HWORD PROCEDURE( LOGICAL_UNIT BYTE, NUMBER WORD )

WRITELN_HWORD PROCEDURE( LOGICAL_UNIT BYTE, NUMBER WORD )

!!!!!!!!!!!!

WRITE_POINTER PROCEDURE( LOGICAL_UNIT BYTE, LOCATION WORD )

WRITELN_POINTER PROCEDURE( LOGICAL_UNIT BYTE, LOCATION WORD )

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

READLN PROCEDURE( LOGICAL_UNIT BYTE, TEXT PBYTE )
       RETURNS  ( OUT_TEXT PBYTE )

READ_HBYTE PROCEDURE( LOGICAL_UNIT BYTE )
           RETURNS  ( NUMBER BYTE )

READ_DBYTE PROCEDURE( LOGICAL_UNIT BYTE )
           RETURNS  ( NUMBER BYTE )

READ_BBYTE PROCEDURE( LOGICAL_UNIT BYTE )
           RETURNS  ( NUMBER BYTE )

READ_LBYTE PROCEDURE( LOGICAL_UNIT BYTE )
           RETURNS  ( NUMBER BYTE )

READ_DINTEGER PROCEDURE( LOGICAL_UNIT BYTE )

READ_DWORD PROCEDURE( LOGICAL_UNIT BYTE )
           RETURNS  ( NUMBER WORD )

READ_HWORD PROCEDURE( LOGICAL_UNIT BYTE )
           RETURNS  ( NUMBER WORD )
```

```
TEST_ENHS PROCEDURE

! This is the testing routine. !

LOCAL
  FIRST_BYTE SECOND_BYTE THIRD_BYTE FOURTH_BYTE   BYTE
  FIRST_INTEGER SECOND_INTEGER THIRD_INTEGER FOURTH_INTEGER  INTEGER
  FIRST_WORD SECOND_WORD THIRD_WORD FOURTH_WORD WORD
  FIRST_PTR SECOND_PTR THIRD_PTR FOURTH_PTR  PBYTE

ENTRY

  WRITE( CONOUT, #'This is a test of the enhancements: WRITE%R' )
  WRITE( CONOUT, #' and this should be on the same line.%R' )
  WRITELN( CONOUT, #' And this should end it.%R' )

  WRITELN( SYSLST, #' This text should go the the system list device.%R' )

  WRITE( SYSLST, #'first block -      %R' )
  WRITE( SYSLST, #'second block -      %R' )
  WRITE( SYSLST, #'third block -      %R' )
  WRITELN( SYSLST, #'%R' )

  FIRST_BYTE := %0
  WRITELN( CONOUT, #'Tests of the write_?byte and writeln_?byte routines.%R' )
  DO
    WRITELN( CONOUT, #'Next value of first_byte%R' )
    WRITE_LBYTE( CONOUT, FIRST_BYTE )
    WRITELN_LBYTE( CONOUT, FIRST_BYTE )
    WRITE_BBYTE( CONOUT, FIRST_BYTE )
    WRITELN_BBYTE( CONOUT, FIRST_BYTE )
    WRITE_DBYTE( CONOUT, FIRST_BYTE )
    WRITELN_DBYTE( CONOUT, FIRST_BYTE )
    WRITE_HBYTE( CONOUT, FIRST_BYTE )
    WRITELN_HBYTE( CONOUT, FIRST_BYTE )
    FIRST_BYTE := FIRST_BYTE + 1
    IF FIRST_BYTE > 40 THEN EXIT FI
  OD
```

```
22      WRITELN( CONOUT, #'End of tests of byte to consol %R' )


23      WRITELN( SYSLST, #'Tests of the write_?byte and writeln_?byte routines.%R' )
        DO
24        WRITELN( SYSLST, #'Next value of first_byte%R' )
25        WRITE_LBYTE( SYSLST, FIRST_BYTE )
26        WRITELN_LBYTE( SYSLST, FIRST_BYTE )
27        WRITE_BBYTE( SYSLST, FIRST_BYTE )
28        WRITELN_BBYTE( SYSLST, FIRST_BYTE )
29        WRITE_DBYTE( SYSLST, FIRST_BYTE )
30        WRITELN_DBYTE( SYSLST, FIRST_BYTE )
31        WRITE_HBYTE( SYSLST, FIRST_BYTE )
32        WRITELN_HBYTE( SYSLST, FIRST_BYTE )
33        FIRST_BYTE := FIRST_BYTE + 1
34        IF FIRST_BYTE = 0 THEN EXIT FI
        OD

35      WRITELN( SYSLST, #'End of tests of byte to syslst %R' )

36      FIRST_INTEGER := -1345
37      WRITELN( CONOUT, #'Test of integer writes.%R' )
        DO
38        WRITE( CONOUT, #'Integer decimal value:%R' )
39        WRITE_DINTEGER( CONOUT, FIRST_INTEGER )
40        WRITE( CONOUT, #' %R' )
41        WRITELN_DINTEGER( CONOUT, FIRST_INTEGER )
42        FIRST_INTEGER := FIRST_INTEGER + 1
43        IF FIRST_INTEGER > 2000 THEN EXIT FI
        OD
44      WRITELN( CONOUT, #' END OF TEST OF INTEGER WRITES%R' )

45      FIRST_INTEGER := -1345
46      WRITELN( SYSLST, #'Test of integer writes.%R' )
        DO
47        WRITE( SYSLST, #',  integer decimal value:%R' )
48        WRITE_DINTEGER( SYSLST, FIRST_INTEGER )
49        WRITE( SYSLST, #' , %R' )
50        WRITELN_DINTEGER( SYSLST, FIRST_INTEGER )
51        FIRST_INTEGER := FIRST_INTEGER + 100
52        IF FIRST_INTEGER > 2000 THEN EXIT FI
        OD
53      WRITELN( SYSLST, #' END OF TEST OF INTEGER WRITES%R' )
```

```
54          FIRST_WORD := 64000
55          WRITELN( CONOUT, #'Test of word writes.%R' )
            DO
56            WRITE( CONOUT, #' Word values: %R' )
57            WRITE_DWORD( CONOUT, FIRST_WORD )
58            WRITE( CONOUT, #' %R' )
59            WRITELN_HWORD( CONOUT, FIRST_WORD )
60            FIRST_WORD := FIRST_WORD - 10
61            IF FIRST_WORD > 64000 THEN EXIT FI
            OD
62          WRITELN( CONOUT, #' END OF WORD TEST%R' )

63          FIRST_WORD := 64000
64          WRITELN( SYSLST, #'Test of word writes.%R' )
            DO
65            WRITE( SYSLST, #' Word values: %R' )
66            WRITE_HWORD( SYSLST, FIRST_WORD )
67            WRITE( SYSLST, #' %R' )
68            WRITELN_DWORD( SYSLST, FIRST_WORD )
69            FIRST_WORD := FIRST_WORD - 1000
70            IF FIRST_WORD > 64000 THEN EXIT FI
            OD
71          WRITELN( SYSLST, #' END OF WORD TEST%R' )

72      END TEST_ENHS

    END TEST_IT
END OF COMPILATION:   0 ERROR(S)      0 WARNING(S)
  713 DATA BYTES    366 Z-CODE BYTES    SYMBOL TABLE 14% FULL
```

Appendix B: Utility Module Listings

The following 17 pages are the assembler listing of the Utility Module routines and listing of two modules of testing routines. The contents of these pages is

```
*heading Module UTILITY








;       2 March 1981 - 1452



;       UTILITY is a set of assembly language routines designed to be called
; by PLZ routines.  These assembly language routines receive parameters
; from and return parameters to the calling PLZ routines via the stack.  The
; pushing and poping of various registers is done to maintain the correct
; logical environment of prameters when passing between the calling routine
; and these assembly routines.
;
;       Module UTILITY contains nine routines.  To be called by a PLZ routine
; the UTILITY routine must be declaired external in the PLZ module contining
; the calling routine.  These nine routines are independent; the module can
; be reduced in size by assemblying only the routines needed (including the
; equates section).  The resulting module (object code) must be linked in with
; the modules of the PLZ routine.
;

```

```
                   *heading IO Port Access Routines



                   ;       IOOUT allows a PLZ routine to output a BYTE value to a specific IO
                   ; port.  This routine should be declaired EXTERNAL as follows.
                   ;
                   ;       IOOUT PROCEDURE( IO_PORT VALUE  BYTE )
                   ;
                   ; This routine is invoked by:
                   ;
                   ;       IOOUT( IO_PORT, VALUE )
                   ;

                           GLOBAL IOOUT

0000  DDE5         IOOUT:          PUSH   IX            ; Save the calling routine's IX
0002  DD210000                     LD     IX,ZERO       ; Clear the IX register.
0006  DD39                         ADD    IX,SP         ; Get the current stack pointer

0008  DD7E04                       LD     A,(IX+4)      ; Load the A register with the VALUE to be written to the port.
000B  DD4E06                       LD     C,(IX+6)      ; Get the address of the IO port value PORT.
000E  ED79                         OUT    (C),A         ; Write the A register to the IO port PORT.

0010  DDE1                         POP    IX            ; Restore the calling program's IX value.
0012  E1                           POP    HL            ; Get the return address.
0013  D1                           POP    DE            ; Deallocate the out parameter VALUE's storage space.
0014  D1                           POP    DE            ; Deallocate the out parameter PORT's storage space.
0015  E9                           JP     (HL)          ; Return to the calling PLZ routine.

                           ; End of routine IOOUT.
```

```
 LOC   OBJ CODE M STMT SOURCE STATEMENT

                   60  *Eject
                   61
                   62
                   63
                   64  ;       IOIN allows a PLZ routine to directly read from an IO port.  The calling
                   65  ; PLZ routine specifies the IO port number; IOIN returns the value of that port.
                   66  ; IOIN is declaired external in the calling PLZ routine's module as follows.
                   67  ;
                   68  ;       IOIN PROCEDURE ( IO_PORT  BYTE )
                   69  ;            RETURNS   ( VALUE    BYTE )
                   70  ;
                   71  ; Routine IOIN is invoked by:
                   72  ;
                   73  ;       VALUE := IOIN( IO_PORT )
                   74  ;
                   75
                   76          GLOBAL IOIN
                   77
0016  DDE5         78  IOIN:           PUSH    IX              ; Save the calling routine's IX value in the stack.
0018  DD210000     79                  LD      IX,0            ; Clear the IX register.
001C  DD39         80                  ADD     IX,SP           ; Get the current value of the Stack Pointer.
                   81
001E  DD4E04       82                  LD      C,(IX+4)        ; Get the address of the IO port value PORT.
0021  ED78         83                  IN      A,(C)           ; Read from the IO port PORT.
0023  DD7706       84                  LD      (IX+6),A        ; Place VALUE in the parameter passing location.
0026  DD360700     85                  LD      (IX+7),ZERO     ; Clear the upper range.
                   86
002A  DDE1         87                  POP     IX              ; Restore the calling PLZ routine's IX value.
002C  E1           88                  POP     HL              ; Get the return address.
002D  D1           89                  POP     DE              ; Deallocate the out parameter PORT's storage space.
002E  E9           90                  JP      (HL)            ; Return to the calling PLZ routine.
                   91
                   92          ; End of routine IOIN.
```

```
 LOC   OBJ CODE M STMT SOURCE STATEMENT

                  93  *heading Memory Access Routines
                  94
                  95
                  96
                  97  ;       Routine MEMSET provides an alternate method for PLZ routines to set
                  98  ; specific memory loctions.  The PLZ routine could set a pointer to the
                  99  ; specific address and then set the location via that pointer.  Routine MEMSET
                 100  ; accomplishes the same task via a subroutine call.  MEMSET is declaired
                 101  ; EXTERNAL to the calling PLZ routine's MODULE as follows.
                 102  ;
                 103  ;       MEMSET PROCEDURE ( LOCATION  WORD,  VALUE  BYTE )
                 104  ;
                 105  ; Routine MEMSET is invoked by:
                 106  ;
                 107  ;       MEMSET( LOCATION, VALUE )
                 108  ;
                 109
                 110          GLOBAL MEMSET
                 111
 002F  DDE5      112  MEMSET:          PUSH    IX          ; Save the calling routine's IX value in the stack.
 0031  DD210000  113                   LD      IX,ZERO     ; Clear the IX register.
 0035  DD39      114                   ADD     IX,SP       ; Get the current value of the stack pointer.
                 115
 0037  DD7E04    116                   LD      A,(IX+4)    ; Get the VALUE to be placed in memory from the stack.
 003A  DD6E06    117                   LD      L,(IX+6)    ;   Get the desired memory LOCATION address, lower half.
 003D  DD6607    118                   LD      H,(IX+7)    ; Upper half of LOCATION memory address.
 0040  77        119                   LD      (HL),A      ; Set the desired memory location to the desired value.
                 120
 0041  DDE1      121                   POP     IX          ; Restore the calling routine's IX value.
 0043  E1        122                   POP     HL          ; Get the return address.
 0044  D1        123                   POP     DE          ; Deallocate the out parameter VALUE's storage space.
 0045  D1        124                   POP     DE          ; Deallocate the out parameter LOCATION's storage space.
 0046  E9        125                   JP      (HL)        ; Return to the calling PLZ routine.
                 126          ; End of routine MEMSET
```

```
                 127  *Eject
                 128
                 129
                 130  ;       Assembly language routine MEMREAD allows a PLZ routine to read the
                 131  ; contents of a specific memory location without resorting to pointers.  This
                 132  ; routine is declaired EXTERNAL in the calling PLZ routine's module as follows.
                 133  ;
                 134  ;       MEMREAD PROCEDURE ( LOCATION  WORD )
                 135  ;               RETURNS   ( VALUE     BYTE )
                 136  ;
                 137  ; MEMREAD is invoked in the PLZ routine by:
                 138  ;
                 139  ;       VALUE := MEMREAD( LOCATION )
                 140  ;
                 141
                 142          GLOBAL MEMREAD
                 143
0047  DDE5       144  MEMREAD:        PUSH   IX              ; Save the calling routine's IX value in the stack.
0049  DD210000   145                  LD     IX,ZERO         ; Clear the IX register.
004D  DD39       146                  ADD    IX,SP           ; Get the current value of the stack pointer.
                 147
004F  DD6E04     148                  LD     L,(IX+4)        ;   Get the desired memory LOCATION address, lower half.
0052  DD6605     149                  LD     H,(IX+5)        ; Upper half of LOCATION.
0055  7E         150                  LD     A,(HL)          ; Get the VALUE from the addressed memory LOCATION.
0056  DD7706     151                  LD     (IX+6),A        ; Place VALUE in the stack for passing back to the PLZ routine.
0059  DD360700   152                  LD     (IX+7),ZERO     ; Clear the upper range.
                 153
005D  DDE1       154                  POP    IX              ; Restore the calling routine's IX value.
005F  E1         155                  POP    HL              ; Get the return address.
0060  D1         156                  POP    DE              ; Deallocate the outparameter LOCATION storage space.
0061  E9         157                  JP     (HL)            ; Return to the calling PLZ routine.
                 158
                 159          ; End of routine MEMREAD.
```

```
                   *heading Interrupt Control Routines




                   ;       DISABLEINT makes it possible for a PLZ language routine to disable the
                   ; Z-80 CPU maskable interrupts.  The CPU should be set for Mode 2 interrupts.
                   ; DISABLEINT is declaired in the EXTERNAL section of the calling routine's
                   ; MODULE as follows.
                   ;
                   ;       DISABLEINT PROCEDURE
                   ;
                   ; Note that DISABLEINT has no input or output parameters. This routine is
                   ; invoked in the calling PLZ routine by:
                   ;
                   ;       DISABLEINT
                   ;

                           GLOBAL DISABLEINT

0062  F3           DISABLEINT:     DI                    ; Disable interrupts.
0063  E1                           POP     HL            ; Get the return address.
0064  E9                           JP      (HL)          ; Return to the calling PLZ routine.

                           ; End of routine DISABLEINT.
```

```
 LOC   OBJ CODE M STMT SOURCE STATEMENT                           ASM 5.8

                    185  *Eject
                    186
                    187
                    188
                    189
                    190  ;       ENABLEINT is the counterpart of DISABLEINT.  This routine enables the
                    191  ; maskable interrupts of the Z-80 CPU.  Please note that if an interrupt is
                    192  ; pending when this routine executes, the interrupt will occur just after the
                    193  ; POP HL instruction; control will not have returned to the calling PLZ routine.
                    194  ; ENABLEINT is declaired EXTERNAL in the calling routine's module as follows.
                    195  ;
                    196  ;       ENABLEINT PROCEDURE
                    197  ;
                    198  ; Note that there are no input or output parameters.  Is routine is invoked by:
                    199  ;
                    200  ;       ENABLEINT
                    201  ;
                    202
                    203         GLOBAL ENABLEINT
                    204
0065   FB           205  ENABLEINT:     EI                       ; Enable interrupts.
0066   E1           206                 POP     HL               ; Get the return address.
0067   E9           207                 JP      (HL)             ; Return to the calling PLZ routine.
                    208
                    209         ; End of routine ENABLEINT.
```

```
 LOC  OBJ CODE M STMT SOURCE STATEMENT                         ASM 5.8

                 210 *heading System DATE Access Routine
                 211 ;       Routine DATE fetches the current system date from memory and returns
                 212 ; the six ASCII characters ( no matter what they are ) to the calling routine.
                 213 ; If the memory location has been scrambled, non-ASCII values are likely.
                 214 ; For this routine to be useful, the system date must be set upon system
                 215 ; boot.  Routine DATE is declaired external in the calling module as follows.
                 216 ;
                 217 ;        DATE PROCEDURE
                 218 ;             RETURNS   ( YEAR1 YEAR0 MONTH1 MONTH0 DAY1 DAY0  BYTE )
                 219 ;
                 220 ; DATE is invoked in the calling PLZ routine by:
                 221 ;
                 222 ;        YEAR1, YEAR0, MONTH1, MONTH0, DAY1, DAY0 := DATE
                 223
                 224          GLOBAL DATE
                 225
0068  DDE5       226 DATE:            PUSH    IX               ; Save the calling routine's IX value in the stack.
006A  DD210000   227                  LD      IX,ZERO          ; Clear the IX register.
006E  DD39       228                  ADD     IX,SP            ; Get the current value of the stack pointer.
                 229
0070  210400     230                  LD      HL, 04H          ; Load the stack offset for output parameters.
0073  3E00       231                  LD      A,ZERO           ; Clear the accumulator.
0075  DDE5       232                  PUSH    IX               ; Load the DE registers with the contents of
0077  D1         233                  POP     DE               ;   of the IX register via a push and pop.
0078  19         234                  ADD     HL,DE            ; Set DE to the location of the first output parameter location.
0079  21AB13     235                  LD      HL,DATE_ADDRESS  ; Set HL to the DATE storage memory location.
007C  010600     236                  LD      BC,6H            ; Set BC to the byte count.
                 237
007F  EDA0       238 DATE_LOOP:       LDI                      ; Load the date character into the output parameter location.
0081  13         239                  INC     DE               ; Adjust for WORD length of output parameter.
0082  B9         240                  CP      C                ; Check if all 6 characters have been loaded.
0083  20FA       241                  JR      NZ,DATE_LOOP     ;   No, continue loading.
                 242                                           ;   Yes, end routine.
                 243
0085  DDE1       244                  POP     IX               ; Restore the calling routine's IX value.
0087  E1         245                  POP     HL               ; Get the return address.
0088  E9         246                  JP      (HL)             ; Return to the calling PLZ program.
                 247
                 248          ; End of routine DATE.
```

```
*heading Memory Manager Access Routines



;       ALLOCATE is a routine to allocate memory via the system memory
; manager.  The calling PLZ routine passes ALLOCATE the size, lower_bound
; and upper_bound of the desired block of memory to be allocated.  These
; values are loaded into the proper registers and the memory manager is
; called.  The memory manager returns the values return_code, available_size,
; and the beginning and ending addresses.  These values are loaded into the
; stack for return to the calling PLZ routine.  See appendix I of the Z80-RIO
; Operating System User's Manual for further information.
;       ALLOCATE is declaired external in the calling routine's module as follows.
;
;       ALLOCATE PROCEDURE ( BLOCK_SIZE_REQUESTED  LOWER_MEMORY_BOUND  UPPER_MEMORY_BOUND   WORD )
;                 RETURNS   ( RETURN_CODE  BYTE,
;                             AVAILABLE_BLOCK_SIZE  BEGINNING_ADDRESS  ENDING_ADDRESS   WORD )
;
;       Note:  If BLOCK_SIZE_REQUESTED is greater than AVAILABLE_BLOCK_SIZE then
; ENDING_ADDRESS will be returned with a value of ZERO and RETURN_CODE will be
; returned with a value of 4AH the RIO return code for insufficient memory.
; AVAILABLE_BLOCK_SIZE will be the number of bytes of the largest unallocated
; block of memory within the specified bounds.  If AVALIABLE_BLOCK_SIZE is
; greater than zero, then BEGINNING_ADDRESS is the address of the largest
; available block of memory.  If AVALIABLE_BLOCK_SIZE is zero, then BEGINNING_
; ADDRESS is returned with a value of zero.
;       ALLOCATE is invoked in the calling PLZ routine by:
;
;       RETURN_CODE, AVAILABLE_BLOCK_SIZE, BEGINNING_ADDRESS, ENDING_ADDRESS :=
;          ALLOCATE( BLOCK_SIZE_REQUESTED, LOWER_MEMORY_BOUND, UPPER_MEMORY_BOUND )
```

```
 LOC  OBJ CODE M STMT SOURCE STATEMENT

                 279 *Eject
                 280
                 281          GLOBAL ALLOCATE
                 282
0089  DDE5       283 ALLOCATE:      PUSH  IX                    ; Save the calling routine's IX value.
008B  DD210000   284                LD    IX,ZERO               ; Clear the IX register.
008F  DD39       285                ADD   IX,SP                 ; Load the IX register with the Stack Pointer.
                 286
0091  3E00       287                LD    A,ALCT_MEMORY         ; Load the allocate command into the A register.
0093  DD6E06     288                LD    L,(IX+06H)            ;   Load the lower memory bound
0096  DD6607     289                LD    H,(IX+07H)            ; address in the HL registers.
0099  DD5E04     290                LD    E,(IX+04H)            ;   Load the high memory bound
009C  DD5605     291                LD    D,(IX+05H)            ; address in the DE registers.
009F  DD4E08     292                LD    C,(IX+08H)            ;  Load the requested block size
00A2  DD4609     293                LD    B,(IX+09H)            ; into the BC registers.
                 294
00A5  CD0914     295                CALL  MEMORY_MANAGER        ; Call the system memory manager.
                 296
00A8  DD770A     297                LD    (IX+0AH),A            ;   Load the memory manager response
00AB  DD360B00   298                LD    (IX+0BH),ZERO         ; into the RETURN_CODE location.
00AF  DD710C     299                LD    (IX+0CH),C            ;   Load the size of the largest available
00B2  DD700D     300                LD    (IX+0DH),B            ; or allocated block into the AVAIL_SIZE location.
                 301
                 302        ; Determine the memory manager's response to the request and form the output appropiately.
                 303
00B5  FE80       304                CP    OPERATION_COMPLETE    ; Was the allocation successful ?
00B7  2810       305                JR    Z,LOAD_OUTPUT         ;   Yes, do not clear the other values.
                 306                                            ;   No, clear the ENDING value and continue checking.
00B9  110000     307                LD    DE,ZERO               ; Return zero as the ending address.
00BC  78         308                LD    A,B                   ;  Load the upper range of SIZE into the A register.
00BD  FE00       309                CP    ZERO                  ;  Is the upper range of SIZE zero ?
00BF  2008       310                JR    NZ,LOAD_OUTPUT        ;    No, there is a block of memory available.
00C1  79         311                LD    A,C                   ;    Yes, check the lower range of SIZE.
00C2  FE00       312                CP    ZERO                  ;      Is the lower range zero ?
00C4  2003       313                JR    NZ,LOAD_OUTPUT        ;        No, there is a block of memory available.
00C6  210000     314                LD    HL,ZERO               ;        Yes, SIZE is zero, there is no memory availabl
                 315                                            ;          Clear the AVAIL_SIZE value.
```

```
 LOC   OBJ CODE M STMT SOURCE STATEMENT

                   316  *eject
                   317
00C9   DD7510      318  LOAD_OUTPUT    LD      (IX+10H),L           ;   Load the beginning address of the block of
00CC   DD7411      319                 LD      (IX+11H),H           ; memory into the BEGINNING location.
00CF   DD730E      320                 LD      (IX+0EH),E           ;   Load the ending address of the block of
00D2   DD720F      321                 LD      (IX+0FH),D           ; memory into the ENDING location.
                   322
                   323       ; Set the stack and register into the proper configuration for return to the calling PLZ routine.
                   324
00D5   DDE1        325                 POP     IX                   ; Restore the calling routine's IX value.
00D7   E1          326                 POP     HL                   ; Get the return address.
00D8   D1          327                 POP     DE                   ; Deallocat storage for UPPER_BOUNDRY.
00D9   D1          328                 POP     DE                   ; Deallocate storage for LOWER_BOUNDRY.
00DA   D1          329                 POP     DE                   ; Deallocate storage for RQST_SIZE.
00DB   E9          330                 JP      (HL)                 ; Return to the calling PLZ routine.
                   331
                   332       ; End of routine ALLOCATE.
                   333
                   334
                   335
                   336
                   337
                   338  ;        Routine DEALLOCATE allows a PLZ language routine to call the system
                   339  ; memory manager and deallocate memory.  This routine is declaired EXTERNAL
                   340  ; in the calling PLZ routine's module as follows.
                   341  ;
                   342  ;        DEALLOCATE PROCEDURE ( BLOCK_SIZE  BEGINNING_ADDRESS  WORD )
                   343  ;                    RETURNS   ( RETURN_CODE  BYTE)
                   344  ;
                   345  ;        There are two possible values for RETURN_CODE.  If the operation was
                   346  ; successful, RETURN_CODE is returned with a value of 80H the RIO return code
                   347  ; for operation complete.  If the memory referenced by BLOCK_SIZE and BEGINNING_
                   348  ; ADDRESS was not a single continuous block of allocated memory, RETURN_CODE will
                   349  ; be returned with the value 43H the RIO return code for memory protect violation.
                   350  ; See Appendix I of the Z80-RIO Operating System User's Manual for further information.
                   351  ;        DEALLOCATE is invoked in the calling PLZ routine by:
                   352  ;
                   353  ;        RETURN_CODE := DEALLOCATE( BLOCK_SIZE, BEGINNING_ADDRESS )
```

```
                  354 *eject
                  355
                  356           GLOBAL DEALLOCATE
                  357
00DC   DDE5       358 DEALLOCATE:     PUSH    IX                    ; Save the calling PLZ routine's IX value.
00DE   DD210000   359                 LD      IX,ZERO               ; Clear the IX register.
00E2   DD39       360                 ADD     IX,SP                 ; Load the stack pointer into the IX register.
                  361
                  362           ; Load the parameters from the PLZ routine into the proper registers for a call to the memory
                  363
00E4   3E01       364                 LD      A,DEALCT_MEMORY       ; Load the deallocate command into the A register.
00E6   DD6E04     365                 LD      L,(IX+4)              ;   Load the BEGINNING address of the memory to
00E9   DD6605     366                 LD      H,(IX+5)              ; be deallocated into the HL registes.
00EC   DD4E06     367                 LD      C,(IX+6)              ;   Load the size of the block of memory to be
00EF   DD4607     368                 LD      B,(IX+7)              ; deallocated into the BC registers.
                  369
00F2   CD0914     370                 CALL    MEMORY_MANAGER        ; Call the memory manager.
                  371
00F5   DD7708     372                 LD      (IX+8),A              ;   Load the RETURN_CODE location with
00F8   DD360900   373                 LD      (IX+9),ZERO           ; the manager's response.
                  374
                  375           ; Put the stack into the proper configruation for return to the calling PLZ routine.
                  376
00FC   DDE1       377                 POP     IX                    ; Restore the calling routine's IX value.
00FE   E1         378                 POP     HL                    ; Get the return address.
00FF   D1         379                 POP     DE                    ; Deallocat the BEGIN location.
0100   D1         380                 POP     DE                    ; Deallocate the DAL_SIZE location.
0101   E9         381                 JP      (HL)                  ; Return to the calling routine.
                  382
                  383           ; End of routine DEALLOCATE.
```

```
                384 *heading EQUATES for UTILITY
                385
                386
                387          ; Equates:  Value of constants.
                388
                389 OPERATION_COMPLETE:     EQU     080H          ; RIO return code for successful operation.
                390 MEMORY_MANAGER:         EQU     1409H         ; Entry point address for system memory manager.
                391 DATE_ADDRESS:           EQU     13ABH         ; Address of the current system date.
                392 ZERO:                   EQU     00            ; Like it says.
                393 ALCT_MEMORY:            EQU     00            ; Command to memory manager for allocation of memory.
                394 DEALCT_MEMORY:          EQU     1             ; Command to memory manager for deallocation of memory.
                395
                396
                397          END            ; End of Module UTILITIES
```

```
SYMBOL  VAL  M DEFN REFS

ALCT_M  0000    393  287
ALLOCA  0089 G  283  281
DATE    0068 G  226  224
DATE_A  13AB    391  235
DATE_L  007F R  238  241
DEALCT  0001    394  364
DEALLO  00DC G  358  356
DISABL  0062 G  180  178
ENABLE  0065 G  205  203
IOIN    0016 G   78   76
IOOUT   0000 G   45   43
LOAD_O  00C9 R  318  305  310  313
MEMORY  1409    390  295  370
MEMREA  0047 G  144  142
MEMSET  002F G  112  110
OPERAT  0080    389  304
ZERO    0000    392   46   85  113  145  152  227  231  284  298  307
                     309  312  314  359  373
```

```
PLZSYS 3.0
         TESTS MODULE


           TYPE

             PBYTE  ^BYTE


           CONSTANT

             CONOUT := %02


           EXTERNAL

             MEMSET PROCEDURE ( LOCATION  WORD,  VALUE  BYTE )


             MEMREAD PROCEDURE ( LOCATION  WORD )
                     RETURNS   ( VALUE     BYTE )

             WRITELN PROCEDURE ( LOGICAL_UNIT  BYTE,  TEXT_POINTER  PBYTE )



           GLOBAL


             TEST1 PROCEDURE

               LOCAL

                 LOCATION  WORD
                 VALUE     BYTE
                 CHARACTER BYTE


               ENTRY
     1           LOCATION := %6000
     2           VALUE    := '0'
                 DO
     3             MEMSET( LOCATION, VALUE )
     4             LOCATION += 1
     5             VALUE    += 1
     6             IF LOCATION >= %6020 THEN EXIT FI
                 OD

                 DO
     7             CHARACTER := MEMREAD( LOCATION )
     8             WRITELN( CONOUT, #CHARACTER)
     9             LOCATION -= 1
    10             IF LOCATION <= %5FFF THEN EXIT FI
                 OD

    11         END TEST1

             END TESTS
END OF COMPILATION:   0 ERROR(S)      0 WARNING(S)
    0 DATA BYTES     48 Z-CODE BYTES    SYMBOL TABLE  2% FULL
```

1

```
ASMTEST MODULE

CONSTANT
     CARRIAGE     := %0D
     LINEFEED     := %0A
     ESCAPE       := %1B
     PRINTER_DATA := %90
     PRINTER_CMD  := %91
     TRANSMITRDY  := %01
     RECEIVERDY   := %02

EXTERNAL
     IOOUT PROCEDURE( PORT VALUE INTEGER )
     IOIN  PROCEDURE( PORT INTEGER )
           RETURNS( VALUE INTEGER )

INTERNAL
     INVAL OUTVAL INDEX  INTEGER

     PORTIN PROCEDURE
            RETURNS ( VALUE INTEGER )

       LOCAL
         STATUS INTEGER

       ENTRY
         DO
           STATUS := IOIN( PRINTER_CMD )
           IF STATUS AND RECEIVERDY = RECEIVERDY THEN EXIT FI
           OD
         VALUE := IOIN( PRINTER_DATA )
         END PORTIN


     PORTOUT PROCEDURE( VALUE INTEGER )

       LOCAL
         STATUS INTEGER

       ENTRY
         DO
           STATUS := IOIN( PRINTER_CMD )
           IF STATUS AND TRANSMITRDY = TRANSMITRDY THEN EXIT FI
           OD
         IOOUT( PRINTER_DATA VALUE )
         END PORTOUT
```

```
GLOBAL
     MAIN PROCEDURE

     ENTRY
       OUTVAL := %00
       DO
         DO
           INVAL := PORTIN
           IF INVAL <> OUTVAL THEN EXIT FI
           OD
         IF INVAL = ESCAPE THEN EXIT FI
         INDEX := 0
         OUTVAL := INVAL
         DO
           PORTOUT( OUTVAL )
           INDEX += 1
           IF INDEX = 5 THEN EXIT FI
           OD
          PORTOUT( CARRIAGE )
          PORTOUT( LINEFEED )
         OD
       END MAIN

END ASMTEST
```

Appendix C:
Time Compairson Using PLZ

As discussed in the introduction to Sampler Module, assembly language was selected primarly for a speed advantage. In Sampler Module (routine COLLECTER) only four assembly language instructions are needed to read a value in from an IO port and check the value against a constant. The alternative was to use a PLZ routine which calls the Utility Module routine IOIN. The listing below is an estimate of the assembly language coding required to accomplish the read and compair with PLZ and the Utility Module IOIN.

```
Lable          Instruction                Cycles          Comment

   Call to IOIN from PLZ Program

LOOP:   LD      HL,RETURN_ADDRESS    4    Save the return address
on
        PUSH    HL                   3      the system stack
        LD      HL, IX+OFFSET        5    Put the IO port number
(input
        PUSH    HL                   3     parameter) on the stack
        JP      IOIN                 3    Go to IOIN

----------------------------------- Utility Module -----------------------------------

IOIN:   PUSH    IX                   4    Save Calling Routine's IX
        LD      IX,ZERO              4    Clear IX register
        ADD     IX,SP                4    Get Offset for Parameters

        LD      C,(IX+4)             5    Get IO Port Number
        IN      A,(C)                3    Call IO Port
        LD      (IX+6),A             5    Load Return Parameter
        LD      (IX+7), ZERO         5    Fill upper byte of return
                                            parameter.
        POP     IX                   4    Get calling routine's IX
        POP     HL                   3    Get Return Address
        POP     DE                   3    Clear Parameter Space
        JP      (HL)                 1    Return to Calling Routine
--------------------------------------------------------------------------------------

   Back to Calling PLZ Routine
```

Lable		Instruction	Cycles	Comment
	POP	IX	4	
	LD	IY,ZERO	4	Set offsets for return
	AD	IV,SP	4	parameters
	LD	r,(IY+d_n)	5	Save return parameter in
	LD	(IX+d_1),r	5	Local AREC.
	LD	r_x,(IX+d_1)	5	Get the returned value
	LD	r_y,(IX+d)	5	Get the check value
	CP	r_x,r_y	5	Compair the values
	JRZ	LINE1	2	They don't match
	LD	(IX+d_3),TRUE	5	They match, logical TRUE
	JR	LINE2	3	Continue
LINE1:	LD	(IX+d_3),FALSE	5	They dont'match, FALSE
LINE2:	LD	r_x,(IX+d_3)	5	Get result of compairson
	LD	r_y,(IX+d_4)	5	Get compairson value
	CP	r_x,r_y	2	Check the values.
	JRZ	BRANCH	2	Data is in, go to next section of code.
	JP	LOOP	3	Data is not ready, cyle through again.
		Sum of Cycles	129:	Data is Ready
			13Ø:	Data is Not Ready

With a clock period of 1.56 µsec per cycle (Ref 2), the estimated times for execution are 2Ø1 µsec when data is ready and 2Ø3 µsec when data is not ready. In compairson, the four lines of assembly language used in routine COLLECTER require only 16 µsec. This substantial difference in time is due to the overhead of parameter passing between routines and the overhead of PLZ's activation records (AREC) used to keep track of parameters. (Ref 6 and 9)

Appendix D: Sampler Module Listings

The following 21 pages are the assembler listing of the Sampler Mod- ule. In addition, there is the 9 page listing of TEST3, a routine used in the initial work with the AIO board. Some of the code in TEST3 is repeated in Sampler Mod- ule. The contents of these pages is

```
LOC   OBJ CODE M STMT SOURCE STATEMENT                    ASM 5.8

                    1  *p 45
```

```
*heading Introduction
;                                                    27 February 1981
;       SAMPLER is a collection of routines which implements a real-time clock
; paced analog to digital data collection routine.  This file is intended to
; be linked in as a MODULE with as PLZ language program.  This collection
; of routines is called by the PLZ program as a procedure.  SAMPLER, the entry
; point to this file, appears as a procedure to the calling PLZ routine.  The
; invocation of these routines is:
;
;        ERROR_CODE, LAST_DATA :=
;            SAMPLER( IO_CHANNEL CTC_MODE TIME_CNST COUNT NUM_SAMPLES FIRST_DATA )
;
; where FIRST_DATA and LAST_DATA are of type PBYTE; IO_CHANNEL, CTC_MODE,
; ERROR_CODE, and TIME_CNST are type BYTE; and COUNT and NUM_SAMPLES
; are type WORD.  FIRST_DATA points to the location of the first sample, lower
; byte, and LAST_DATA points to the location of the last sample, lower byte.
; IO_CHANNEL is the number of the analog to digital converter channel to be
; used, 0 to 15.  TIMER_MODE is an 8 bit value in the format of a CTC command.
; This parameter is used to program the sampling period and command the CTC to
; an interrupting timer mode. TIME_CNST is the remaining information needed
; to program the timing period of the CTC.  For long sampling periods (>.001
; sec typically ) the parameter COUNT_VALUE supplies the number of CTC interrupts
; which must pass befor the interrupt service routine is called.  NUM_SAMPLES
; is the number of analog to digital conversions that will be performed.
;
;      SAMPLER must be declaired EXTERNAL in the calling PLZ routine:
;
; EXTERNAL
;      SAMPLER PROCEDURE ( IO_CHANNEL CTC_MODE TIME_CNST BYTE,
;                          COUNT NUM_SAMPLES WORD,
;                          FIRST_DATA PBYTE )
;              RETURNS   ( ERROR_CODE BYTE, LAST_DATA PBYTE )
;
;       The PUSHing and POPing in this routine is the overhead of a PLZ procedure
; call.  The user is u-r-g-e-d to consult the PLZ reference manual.  The basic
; format of this real-time-clock / interrupt-service-routine can be adapted
; to many other applications.  The real time clock has a period varying from
; 6 microseconds to nearly 30 minutes.  In SAMPLER the shortest period used
; is 50 microseconds.  The period of the real time clock is determined by
; the calling routine.
```

```
 LOC   OBJ CODE M STMT SOURCE STATEMENT

                   42  *heading Main routine of SAMPLER
                   43  ;|||||||||| global routine ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
                   44
                   45          GLOBAL         SAMPLER
                   46
                   47
0000  DDE5         48  SAMPLER:        PUSH   IX              ; Save the calling PLZ routine's IX value.
0002  DD210000     49                  LD     IX,0            ; Clear the IX register.
0006  DD39         50                  ADD    IX,SP           ; Get the current stack pointer for parameter passing.
                   51
0008  CD2B00    R  52                  CALL   VALIDATE        ; Validate the correctness of the input to SAMPLER.
000B  201A         53                  JR     NZ,END_SAMPLER  ; If input is invalid, end the routine.
                   54
000D  CD4A00    R  55                  CALL   ATODINIT        ; Initialize the analog to digital (A to D) converter.
                   56
0010  CD5F00    R  57                  CALL   CTC_PROGRAM     ; Program the CTC to the desired mode.
                   58
0013  CD6900    R  59                  CALL   INT_SET_UP      ; Determine the proper interrupt service routine and
                   60                                         ; set all interrupt response parameters.
                   61
0016  CD9400    R  62                  CALL   INIT_COLLECTER  ; Ready the interrupt paced data collection routine.
                   63
0019  CDA100    R  64  ALL_SET:        CALL   USER_READY?     ; Wait for user command to begin.
001C  2006         65                  JR     NZ,DONE         ; If the user aborted turn everything off and return.
                   66
001E  CDF900    R  67  BEGIN:          CALL   START_TIMER     ; Command the CTC to begin timing and interrupting.
                   68
0021  CDFF00    R  69  MAIN_ROUTINE:   CALL   COLLECTER       ; Enter the data collection routine.
                   70
0024  CD1E01    R  71  DONE:           CALL   CTC_OFF         ; All data has been collected.  Disable CTC interrupts.
                   72
0027  CD4401    R  73  END_SAMPLER:    CALL   DEALLOCATE      ; Prepare for return to the calling routine.
                   74
002A  E9           75                  JP     (HL)            ; Return to the calling PLZ routine.
                   76
                   77                                         ; End of routine SAMPLER.
```

```
Routine VALIDATE              SAMPLER       810227.1228       PAGE   4
 LOC   OBJ CODE M STMT SOURCE STATEMENT                         ASM 5.8

                    78 *heading Routine VALIDATE
                    79
                    80
                    81 ;|||||||||| internal routines |||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
                    82
                    83
                    84 ;     VALIDATE checks the input parameters IO_CHANNEL and CTC_MODE aginst their defined ranges and
                    85 ; IO_CHANNEL is defined as a byte, ranging from 0 to 15 decimal.  If IO_CHANNEL is beyond this ran
                    86 ; ERROR_CODE is returned with the value CHANNEL_INVALID.  Input CTC_MODE is restricted to two valu
                    87 ; FAST_MODE and SLOW_MODE.  If CTC_MODE is neither of these values ERROR_CODE is set to MODE_INVAL
                    88 ; For both input checks if all is ok then the Z flag is set upon return to the main routine.
                    89
002B   DD7E0E       90 VALIDATE:     LD      A,(IX+IO_CHANNEL)       ; Get the input channel number.
002E   E6F0         91               AND     UPPER_FOUR              ; Are any of the upper 4 bits ones. ?
0030   2807         92               JR      Z, CHECK_MODE           ;  No, continue checking.
0032   3ECA         93               LD      A,CHANNEL_INVALID       ;  Yes, channel number is beyond defined range.  End
0034   DD7710       94               LD      (IX+ERROR_CODE),A       ;      Load the error code for return.
0037   1810         95               JR      END_VALIDATE            ; Prepare for return.
                    96
0039   DD7E0C       97 CHECK_MODE:   LD      A,(IX+CTC_MODE)         ; Get the CTC command word.
003C   FE87         98               CP      FAST_MODE               ; Is it fast mode ?
003E   2809         99               JR      Z, END_VALIDATE         ;   Yes, CTC_MODE is valid, return.
                   100                                               ;   No, check some more.
0040   FEA7        101               CP      SLOW_MODE               ; Is it slow mode ?
0042   2805        102               JR      Z, END_VALIDATE         ;   Yes, CTC_MODE is valid.
0044   3ECC        103               LD      A,MODE_INVALID          ;   No, CTC_MODE is invalid.
0046   DD7710      104               LD      (IX+ERROR_CODE),A       ;      Load the error_code for return.
                   105
0049   C9          106 END_VALIDATE: RET                             ; Return to the main routine.
                   107
                   108                                               ; End of routine VALIDATE
```

```
                   109  *heading Routine ATODINIT
                   110
                   111
                   112
                   113
                   114
                   115  ;     Routine ATODINIT initializes the A to D converter of the
                   116  ;AIO board into a polled mode.  This routine needs to be run after any
                   117  ;system rest action.  ATODINIT takes 52.90 microseconds, not inculding
                   118  ;the call to this subroutine.
                   119
                   120
004A   F5          121  ATODINIT:     PUSH   AF                   ;Save the AF registers in the stack.
004B   F3          122                DI                          ;Disable system interupts.
                   123
004C   3E4F        124                LD     A,InMode             ;Load the A register with the PIO Mode 1 command.
004E   D322        125                OUT    (CMD_A_PORT),A       ;Write the command to the Port A command register.
0050   D323        126                OUT    (CMD_B_PORT),A       ;Write the command to the Port B command register.
                   127
0052   3E07        128                LD     A,INTDisable         ;Load the PIO interupt disable command into A.
0054   D322        129                OUT    (CMD_A_PORT),A       ;Write the command to the Port A command register.
0056   D323        130                OUT    (CMD_B_PORT),A       ;Write the command to the B Port command register.
                   131
0058   DB20        132                IN     A,(DataLower)        ;Clear the lower data register.
005A   DB21        133                IN     A,(DataUpper)        ;Clear the upper data register.
                   134
005C   FB          135                EI                          ;Enable system interupts.
005D   F1          136                POP    AF                   ;Restore the AF register values.
005E   C9          137                RET                         ;Return to main routine.
                   138
                   139                                            ; End of routine ATODINIT
```

```
                      140  *heading Routine CTC_PROGRAM
                      141
                      142
                      143
                      144
                      145  ;       CTC_PROGRAM loads the CTC mode select register with the value specified
                      146  ; by the calling routine (CTC_MODE) and the CTC interrupt vector register.  The
                      147  ; time constant is loaded by another routine.
                      148
 005F  DD7E0C         149  CTC_PROGRAM:     LD      A,(IX+CTC_MODE) ; Get the mode command parameter.
 0062  D384           150                   OUT     (CTC1_CMD),A        ; Command CTC1 to the desired mode.
                      151
 0064  3E40           152                   LD      A,INT_VECTOR        ; Load the CTC portion of the interrupt vector.
 0066  D384           153                   OUT     (CTC1_CMD),A        ; Load the vector into the CTC.
                      154
 0068  C9             155                   RET                         ; Return to the main program.
                      156
                      157                                               ; End of routine CTC_PROGRAM
```

```
                  158 *heading Routine INI_SET_UP
                  159
                  160
                  161
                  162
                  163 ;       INT_SET_UP sets all parameters of the interrupt servce routin.  Since
                  164 ; the range of sampling times requires the two service routines, INT SET_UP
                  165 ; determines which is applicable and loads the address of that routine into the
                  166 ; interrupt jump table location for CTC1 channel 0.  Additionall INIT_SET_UP
                  167 ; loads the channel number of the desired A to D converter and the
                  168 ; down counter count (if one exists) into the alternate register set for
                  169 ; use by the interrupt service routine.
                  170
0069  08          171 INT_SET_UP:   EX    AF,AF'               ; Exchange the AF registers.
006A  DD7E0E      172               LD    A,(IX+IO_CHANNEL)    ; Load the A register with the analog channel number.
006D  08          173               EX    AF,AF'               ; Restore the AF registers.
                  174
006E  DD7E08      175               LD    A,(IX+COUNT)         ; Get the count_down counter value.
0071  FE00        176               CP    ZERO                 ; Is the low byte of the counter value zero ?
0073  200E        177               JR    NZ,TIME_N_COUNT      ;   No, use the timer-counter interrupter.
0075  DD7E09      178               LD    A,(IX+COUNT+1)       ;   Yes, check the high byte..
0078  FE00        179               CP    ZERO                 ;     Is the high byte of the counter value zero ?
007A  2007        180               JR    NZ,TIME_N_COUNT      ;       No, use the timer-counter interrupter.
                  181                                          ;       Yes, the counter value is zero. Use the
                  182                                          ;       timer-only version of the interrupter.
```

```
 LOC   OBJ CODE M STMT SOURCE STATEMENT

                   183 *heading INT_SET_UP continued
                   184
                   185
                   186 ;       The sampling period is less than .001 seconds.  A counter is not necessary to generate this
                   187 ; sampling period.  Routine TO_SAMPLE will be the interrupt service routine.
                   188
007C  212501    R  189 NO_COUNT:      LD      HL,TO_SAMPLE           ; Load HL with the address of the routine TO_SAMPLE.
007F  224014       190                LD      (INT_JUMP_TABLE),HL    ; Load the address into the interrupt jump table
                   191                                               ; location for CTC1 channel 0.
0082  C9           192                RET                            ; Initializations complete for the timer interrupt
                   193                                               ; service routine.  Return to the main program.
                   194
                   195
                   196
                   197
                   198 ;       The sampling period is greater then .001 seconds.  Both a timer and a counter will
                   199 ; be used.  The interrupt service routine will be TC_SAMPLE.
                   200
0083  212B01    R  201 TIME_N_COUNT:  LD      HL,TC_SAMPLE           ; Get the starting address of the routine TC_SAMPLE.
0086  224014       202                LD      (INT_JUMP_TABLE),HL    ; Place the address in the interrupt jump table
                   203                                               ; location for CTC1 channel 0.
                   204
0089  D9           205                EXX                            ; Exchange the BC, DE, and HL register pairs.
008A  DD4608       206                LD      B,(IX+COUNT)           ; Load the down_counter value, low byte, into B & D
008D  50           207                LD      D,B                    ;  registers, B for operation, D for reset.
008E  DD4E09       208                LD      C,(IX+COUNT+1)         ; Load the down_counter value, high byte, into C & E
0091  59           209                LD      E,C                    ; registers, C for operations, E for reset.
0092  D9           210                EXX                            ; Restore the BC, DE, and HL registers.
                   211
0093  C9           212                RET                            ; Initializations complete for timer/counter interrupt
                   213                                               ; service routine.  Return to the main program.
                   214
                   215                                               ; End of routine INT_SET_UP
```

```
LOC   OBJ CODE M STMT SOURCE STATEMENT

                 216  *heading Routine INIT_COLLECTOR
                 217
                 218
                 219
                 220
                 221  ;       INIT_COLLECTER loads the parameters for the user, interrupt paced,
                 222  ; routine into the proper registers.
                 223
0094  DD5604     224  INIT_COLLECTER: LD   D,(IX+FIRST_DATA)       ; Load the address for the first data word storage, lower
0097  DD5E05     225                  LD   E,(IX+FIRST_DATA+1)     ; byte in D, upper byte in E.
                 226                                               ; location into the DE register pair.
009A  DD4606     227                  LD   B,(IX+NUM_SAMPLES)      ; Load the number of data samples to be taken,
009D  DD4E07     228                  LD   C,(IX+NUM_SAMPLES+1)    ; lower in B, upper byte in C.
                 229
00A0  C9         230                  RET                          ; Return to the main program.
                 231
                 232                                               ; End of routine INIT_COLLECTER
```

```
Routine USER_READY?          SAMPLER        810227.1228       PAGE 10
 LOC   OBJ CODE M STMT SOURCE STATEMENT                        ASM 5.8

                   233  *heading Routine USER_READY?
                   234
                   235
                   236
                   237
                   238  ;       USER_READY? writes a message to the console and then reads a character
                   239  ; from the console.  This input character signifies that the user is ready for
                   240  ; the process to begin.
                   241
00A1   3E00        242  USER_READY?:    LD     A,FALSE               ; Load the all_is_ok error_code into the A register.
00A3   DD7710      243                  LD     (IX+ERROR_CODE),A     ; Load the error code into the return location.
                   244
00A6   FD215601 R  245                  LD     IY,A_VECTOR           ; Load the pointer to the system call vector.
                   246
00AA   3E02        247                  LD     A,CONOUT              ; Load the logical unit number for console
00AC   325601   R  248                  LD     (A_LOGICAL_UNIT),A    ; output into the system call vector.
00AF   3E10        249                  LD     A,WRITELN             ; Load the request code for write line
00B1   325701   R  250                  LD     (A_REQUEST_CODE),A    ; into the system call vector.
00B4   216101   R  251                  LD     HL,MESSAGE            ; Load the pointer to the message into the data
00B7   225801   R  252                  LD     (A_DATA_TRANS),HL     ; transfer position of the system call vector.
00BA   3E21        253                  LD     A,L_MESSAGE           ; Load the length of the messageinto
00BC   325A01   R  254                  LD     (A_BYTE_COUNT),A      ; the byte count of the call vector.
00BF   21CB00   R  255                  LD     HL,SET?               ; Load the address of SET? into the
00C2   225C01   R  256                  LD     (A_RETURN),HL         ; OK return address and the error return
00C5   225E01   R  257                  LD     (A_ERR_RETURN),HL     ; address of the system call vector.
00C8   CD0314      258                  CALL   SYSTEM                ; Call the operating system.
```

```
 LOC   OBJ CODE M STMT SOURCE STATEMENT

                  259  *heading USER_READY? continued
                  260
                  261
                  262
                  263
00CB  3E01        264  SET?:     LD    A,CONIN                ; Load the logical unit for console
00CD  325601   R  265            LD    (A_LOGICAL_UNIT),A     ; input into the system call vector.
00D0  3E0C        266            LD    A,READIN               ; Load the request code for read line
00D2  325701   R  267            LD    (A_REQUEST_CODE),A     ; into the system call vector.
00D5  218201   R  268            LD    HL,RTN_MESS            ; Load the address for the return message
00D8  225801   R  269            LD    (A_DATA_TRANS),HL      ; into the data transfer pointer location.
00DB  3E02        270            LD    A,2                    ; Load the number of bytes to be read
00DD  325A01   R  271            LD    (A_BYTE_COUNT),A       ; into byte count field of the vector.
00E0  21EC00   R  272            LD    HL,GO                  ; Load the address of GO into the OK return
00E3  225C01   R  273            LD    (A_RETURN),HL          ; address field and the error return address
00E6  225E01   R  274            LD    (A_ERR_RETURN),HL      ; field of the system call vector.
00E9  CD0314      275            CALL  SYSTEM                 ; Call the operating system
                  276
00EC  3A5C01   R  277  GO:       LD    A,(A_RETURN)           ; Get the user input character.
00EF  FE59        278            CP    Y_ASCII                ; Is it a 'Y' ?
00F1  C8          279            RET   Z                      ;   Yes, return to the main routine.
00F2  3EAB        280            LD    A,ABORT                ;   No, load the ABORT error code into
00F4  DD7710      281            LD    (IX+ERROR_CODE),A      ;   the error code return location.
00F7  C9          282            RET                          ; Return to the main routine.
00F8  C9          283            RET                          ; Return to the calling routine.
                  284
                  285                                         ; End of routine USER_READY?
```

```
                   286  *heading Routine START_TIMER
                   287
                   288
                   289
                   290
                   291  ;       START_TIMER outputs the time constant word to the CTC.  Since the CTC has been previously
                   292  ; prepaired to expect this word, receipt of the time constant causes the CTC to begin timing and
                   293  ; interrupting.
                   294
00F9  DD7E0A       295  START_TIMER:   LD      A,(IX+TIME_CNST)        ; Get the timer time_constant parameter.
00FC  D384         296                 OUT     (CTC1_CMD),A            ; Load the CTC with the time constant.
00FE  C9           297                 RET                             ; Return to the main program.
                   298
                   299                                                 ; End of routine START_TIMER
```

```
                    300  *heading Routine COLLECTER
                    301
                    302
                    303
                    304  ;      COLLECTER is the interrupt paced data collection routine.  This routine polls the
                    305  ; status of the A to D converter.  When the status shows a new data word is available the
                    306  ; routine reads in the 12 bits of converted data and stores it in memory.  When all data
                    307  ; has been collected, the BC register will be zero and the program will return to the
                    308  ; calling routine.
                    309
                    310  COLLECTER:
00FF  215401   R    311  READY?:      LD    HL,L_BUFFER        ; Load the address of the transfer buffer.
0102  DB29          312               IN    A,(AtoDStatus)     ; Is a new data word available ?
0104  CB47          313               BIT   0,A                ;   (check the status bit.)
0106  28F7          314               JR    Z, READY?          ;  No, check again.
                    315                                        ;  Yes, a new data word is ready.
0108  03            316  DATE_READY:  INC   BC                 ; Offset the sample count by 1 to account for
                    317                                        ; transfer of both bytes.
0109  DB20          318               IN    A,(DataLower)      ; Read the lower 8 bits of the data word.
010B  325401   R    319               LD    (L_BUFFER),A       ; Save the data in the low byte buffer
010E  EDA0          320               LDI                      ; Save the data, lower byte in the location
                    321                                        ; addressed by (DE), decrement the sample count
                    322                                        ; (BC), now returned to baseline.
0110  DB21          323               IN    A,(DataUpper)      ; Read the upper 4 bits of the data word.
0112  325501   R    324               LD    (H_BUFFER),A       ; Place the data in the high byte buffer.
0115  EDA0          325               LDI                      ; Save the data, upper byte, in the
                    326                                        ; location addressed by (DE), decrement the
                    327                                        ; sample count in the BC register.
                    328
0117  E21C01   R    329  END_CHECK:   JP    PO,FINISHED        ; If all samples have been read then end the routine.
011A  18E3          330               JR    READY?             ; transfer buffer.  There are more samples to be read.
                    331
011C  1B            332  FINISHED:    DEC   DE                 ; Decrement the DE register to point to the
                    333                                        ;   lower byte of the last stored data word.
011D  C9            334               RET                      ; Return to the main program.
                    335
                    336                                        ; End of routine COLLECTOR
```

```
LOC   OBJ CODE M STMT SOURCE STATEMENT

                 337 *heading Routine CTC_OFF
                 338 ;       CTC_OFF turns off the CTC interrupts & the timing function of the CTC.
                 339
                 340
                 341
                 342
                 343
011E  F3         344 CTC_OFF:        DI                          ; Disable interrupts
011F  3E78       345                 LD      A,CMD_CTC_OFF       ; Get the CTC reset command.
0121  D384       346                 OUT     (CTC1_CMD),A        ; Turn off the CTC, interrupts are disabled.
0123  FB         347                 EI                          ; Enable system interrupts.
0124  C9         348                 RET                         ; Return to the calling routine.
                 349                                             ; End of routine CTC_OFF
```

```
                 *heading Interrupt Service Routines
                 ;|||||||||| interrupt service routines ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||

                 ;        TO_SAMPLE is the interrupt service routine for sample periods less than .001 seconds.
                 ; No counter is used.  Upon each timer interrupt an A to D conversion is initiated by writing
                 ; the desired channel number to the AIO analog input channel select register.

0125  08         TO_SAMPLE:     EX     AF,AF'                  ; Get the alternate AF registers.
0126  D328                      OUT    (Channel_Select),A      ; Initiate a A to D conversion on the desired channel.
0128  08                        EX     AF,AF'                  ; Restore the AF registers.
0129  ED4D                      RETI                           ; Return from interrupt.

                                                               ; End of routine TO_SAMPLE

                 ;        TC_SAMPLE is the interrupt service routine for sample periods greater than .001 seconds.  A time
                 ; the CTC, and a counter are used to form the sampling period.  Each timer interrupt decrements th
                 ; counter value by one.  When the down counter reaches zero an A to D conversion is initiated by w
                 ; the desired channel number to the AIO analog input channel select register.

012B  08         TC_SAMPLE:     EX     AF,AF'                  ; Get the alternate AF registers.
012C  D9                        EXX                            ; Get the alternate BC, DE, and HL registers.
012D  05                        DEC    B                       ; Decrement the down_counter value, low byte.
012E  2804                      JR     Z,LOWER_ZERO            ; If the lower byte is zero branch
0130  D9                        EXX                            ;   Otherwise, restore the primary
0131  08                        EX     AF,AF'                  ;   registers and return from
0132  ED4D                      RETI                           ;   interrupt.

0134  0D         LOWER_ZERO:    DEC    C                       ; Decrement the down_counter value, high byte.
0135  2805                      JR     Z,COUNTER_ZERO          ; If count is complete go to DONE.
0137  42                        LD     B,D                     ;    Otherwise, reset the low byte counter value,
0138  D9                        EXX                            ;    restore the primary registers,
0139  08                        EX     AF,AF'                  ;    and return from
013A  ED4D                      RETI                           ;    interrupt.

013C  D328       COUNTER_ZERO:  OUT    (Channel_Select),A      ; Initiate an A to D conversion on the desire channel.
013E  4B                        LD     C,E                     ; Reset the down_counter, high byte.
013F  42                        LD     B,D                     ; Reset the down_counter, low byte.
0140  D9                        EXX                            ; Restore the BC, DE, and HL primary registers.
0141  08                        EX     AF,AF'                  ; Restore the AF primary registers.
0142  ED4D                      RETI                           ; Return from interrupt.
                                                               ; End of routine TC_SAMPLE.
```

```
 LOC  OBJ CODE M STMT SOURCE STATEMENT

                 391  *heading Routine DEALLOCATE
                 392
                 393
                 394
                 395  ;       DEALLOCATE stores the output parameters and deallocates storage of the input parameters
                 396  ; before returning to the calling PLZ routine.
                 397
0144  DD7212     398  DEALLOCATE:   LD    (IX+LAST_DATA),D     ; Load the value of the return parameter
0147  DD7313     399                LD    (IX+LAST_DATA+1),E   ;LAST_DATA which points to the buffer location.
014A  DDE1       400                POP   IX                   ; Get the calling routine's IX register value.
014C  E1         401                POP   HL                   ; Get the return address.
014D  D1         402                POP   DE                   ; Deallocate storage for the passed parameters.
014E  D1         403                POP   DE                   ; "
014F  D1         404                POP   DE                   ; "
0150  D1         405                POP   DE                   ; "
0151  D1         406                POP   DE                   ; "
0152  D1         407                POP   DE                   ; "
                 408
0153  C9         409                RET                        ; Return to the main routine.
                 410
                 411                                           ; End of routine DEALLOCATE.
```

```
                 412 *heading Data Storage
                 413
                 414
                 415 ;|||||||||| storage |||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
                 416
                 417
0154  00         418 L_BUFFER:          DEFB   0            ; Buffer for data transfer, low byte.
0155  00         419 H_BUFFER:          DEFB   0            ; Buffer for data transfer, high byte.
                 420
                 421 A_VECTOR:                              ; Storage for the operating system call vector.
0156  00         422 A_LOGICAL_UNIT:    DEFB   0            ; Storage for the logical unit.
0157  00         423 A_REQUEST_CODE:    DEFB   0            ; Storage for the request code.
0158  0000       424 A_DATA_TRANS:      DEFW   0            ; Storage of the pointer to the data transfer location.
015A  0000       425 A_BYTE_COUNT:      DEFW   0            ; Storage of the length of data transfer.
015C  0000       426 A_RETURN:          DEFW   0            ; Storage of the ok return address.
015E  0000       427 A_ERR_RETURN:      DEFW   0            ; Storage of the error return address.
0160  00         428 A_COMP_CODE:       DEFB   0            ; Storage for the completion code.
                 429
                 430 ;                Storage of messages
                 431
0161  436F6C6C   432 MESSAGE:           DEFM   'Collection system ready.  Begin ?'
                 433 L_MESS:            EQU    $-MESSAGE       ; Define the message length.
0182  41414141   434 RTN_MESSAGE:       DEFM   'AAAAAAAAAAAAAAAAAAAAAAAAAA'
```

```
*heading Label Equates
;|||||||||| equates |||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
IO_CHANNEL:       EQU   0EH    ; Offset for the input parameter channel number.
CTC_MODE:         EQU   0CH    ; Offset for input parameter CTC_MODE.
TIME_CNST:        EQU   0AH    ; Offset for the input parameter CTC time constant.
COUNT:            EQU   08H    ; Offset for the input parameter down_counter value.
NUM_SAMPLES:      EQU   06H    ; Offset for the input parameter number of samples.
FIRST_DATA:       EQU   04H    ; Offset for the input parameter pointed to the first
ERROR_CODE:       EQU   10H    ; Offset for the output parameter ERROR_CODE
                               ;  storage location.
LAST_DATA:        EQU   12H    ; Offset for the output parameter pointing to the
                               ;   last data storage location.
CMD_A_PORT:       EQU   22H    ; Address of PIO Port A command register.
CMD_B_PORT:       EQU   23H    ; Address of PIO Port B command register.
DataLower:        EQU   20H    ; Address of AtoD lower range data register.
DataUpper:        EQU   21H    ; Address of AtoD upper range data register.
Channel_Select:   EQU   28H    ; Address of AIO analog input channel select register.
AtoDStatus:       EQU   29H    ; Address of AIO analog input status register.
CTC1_CMD:         EQU   84H    ; IO port address for CTC1 channel 0.
InMode:           EQU   4FH    ; PIO command for input mode.
INTDisable:       EQU   07H    ; PIO command to disable interrupts.
CMD_CTC_OFF:      EQU   78H    ; CTC command to disable interrupts and hault.
FAST_MODE         EQU   87H    ; CTC command for interrupting timer with 16 as prescaler.
SLOW_MODE         EQU   0A7H   ; CTC command for interrupting timer with 256 as prescaler.
```

```
*heading Lable Equates continued

ZERO:                EQU     0H          ; Zero, that's all.
UPPER_FOUR           EQU     0F0H        ; 11110000 in binary.

CONIN                EQU     01H         ; Logical unit number for console input.
CONOUT:              EQU     02H         ; Logical unit number for console output.
SYSLST:              EQU     03H         ; Logical unit number for the system listing device.
INT_VECTOR:          EQU     40H         ; CTC portion of the interrupt vector.
SYSTEM:              EQU     1403H       ; Operating system entry point.
INT_JUMP_TABLE:      EQU     1440H       ; Address of the system interrupt jump table location
                                         ;  for CTC-1, channel 0.
WRITELN:             EQU     10H         ; Request code for writeln.
READLN:              EQU     0CH         ; Request code for read line.

FALSE:               EQU     0H          ; ERROR_CODE value for all_is_ok.
ABORT:               EQU     0ABH        ; ERROR_CODE value for user abort.
FATAL:               EQU     0FFH        ; ERROR_CODE for fatal errors.
CHANNEL_INVALID:     EQU     0CAH        ; ERROR_CODE value for invalid channel number.
MODE_INVALID:        EQU     0CCH        ; ERROR_CODE value for invalid CTC_MODE.

Y_ASCII:             EQU     59H         ; ASCII for 'Y'.
```

```
SYMBOL  VAL M DEFN REFS

ABORT  00AB    483  280
ALL_SE 0019 R   64
ATODIN 004A R  121   55
A_BYTE 015A R  425  254  271
A_COMP 0160 R  428
A_DATA 0158 R  424  252  269
A_ERR_ 015E R  427  257  274
A_LOGI 0156 R  422  248  265
A_REQU 0157 R  423  250  267
A_RETU 015C R  426  256  273  277
A_VECT 0156 R  421  245
AtoDSt 0029    458  312
BEGIN  001E R   67
CHANNE 00CA    485   93
CHECK_ 0039 R   97   92
CMD_A_ 0022    453  125  129
CMD_B_ 0023    454  126  130
CMD_CT 0078    462  345
COLLEC 00FF R  310   69
CONIN  0001    472  264
CONOUT 0002    473  247
COUNT  0008    444  175  178  206  208
COUNTE 013C R  384  378
CTC1_C 0084    459  150  153  296  346
CTC_MO 000C    442   97  149
CTC_OF 011E R  344   71
CTC_PR 005F R  149   57
Channe 0028    457  358  384
DATE_R 0108 R  316
DEALLO 0144 R  398   73
DONE   0024 R   71   65
DataLo 0020    455  132  318
DataUp 0021    456  133  323
END_CH 0117 R  329
END_SA 0027 R   73   53
END_VA 0049 R  106   95   99  102
ERROR_ 0010    448   94  104  243  281
FALSE  0000    482  242
FAST_M 0087    463   98
FATAL  00FE    484
FINISH 011C R  332  329
FIRST_ 0004    446  224  225
GO     00EC R  277  272
H_BUFF 0155 R  419  324
INIT_C 0094 R  224   62
```

SYMBOL	VAL	M	DEFN	REFS	
INIDis	0007		461	128	
INT_JU	1440		477	190	202
INT_SE	0069	R	171	59	
INT_VE	0040		475	152	
IO_CHA	000E		441	90	172
InMode	004F		460	124	
LAST_D	0012		450	398	399
LOWER_	0134	R	377	372	
L_BUFF	0154	R	418	311	319
L_MESS	0021		433	253	
MAIN_R	0021	R	69		
MESSAG	0161	R	432	251	433
MODE_I	000C		486	103	
NO_COU	007C	R	189		
NUM_SA	0006		445	227	228
READLN	000C		480	266	
READY?	00FF	R	311	314	330
RIN_ME	0182	R	434	268	
SAMPLE	0000	G	48	45	
SET?	00CB	R	264	255	
SLOW_M	00A7		464	101	
START_	00F9	R	295	67	
SYSLST	0003		474		
SYSTEM	1403		476	258	275
TC_SAM	012B	R	369	201	
TIME_C	000A		443	295	
TIME_N	0083	R	201	177	180
TO_SAM	0125	R	357	189	
UPPER_	00F0		470	91	
USER_R	00A1	R	242	64	
VALIDA	002B	R	90	52	
WRITEL	0010		479	249	
Y_ASCI	0059		488	278	
ZERO	0000		469	176	179

```
                                   TEST3          810322            PAGE   1
 LOC   OBJ CODE M STMT SOURCE STATEMENT                              ASM 5.8

                     1
                     2
0000   00            3  TEST3:          NOP
                     4
                     5
                     6        GLOBAL          TEST3,ATODINIT,LoopStart,READY?,DataIsReady,OUTDA,CHECKEND
                     7
                     8
                     9
```

```
LOC   OBJ CODE M STMT SOURCE STATEMENT

                 10  *Eject
                 11  ;begin: CLOCKINIT=====================================================================
                 12  ;
                 13  ;       Section CLOCKINIT initializes and starts the realtime clock of the system.
                 14  ;CTC zero, channel zero, on the SIB board is used in the timer mode.  CLOCKINIT
                 15  ;commands the CTC into timer mode and loads the CTC interrupt vector.  From this
                 16  ;point on TOTAL is in control of the interrupt response of the system.  When ever
                 17  ;the CTC time reaches zero it will interrupt the CPU and send the CTC interupt vector
                 18  ;to the CPU.  The CPU will use its interrupt vector as the high eight bits and the
                 19  ;CTC interrupt vector as the lower eight bits of the address of the interupt service
                 20  ;routine.  For this program that routine is called RTCLOCK for Real Time Clock.
                 21  ;CLOCKINIT ends by writing the selected time constant to the CTC time constant register.
                 22  ;This action starts the CTC.  The period of time measured by this clock routine is
                 23  ;dependent upon this time constant.  The period of time is:
                 24  ;
                 25  ;               Period = (System Clock) * (Prescaler) * (Time Constant).
                 26  ;
                 27  ;                        where the system clock period is .4069010416 microseconds.
                 28  ;                              the prescaler is 256.  ( It can also be 16 if desired )
                 29  ;                              and the time constant is user selected.
                 30  ;Once for each CTC timer period the interrupt service routine will be called.
                 31
                 32          GLOBAL          CLOCKINIT
                 33
0001  F3         34  CLOCKINIT:      DI                              ;Disable system interrupts.
                 35
0002  216200   R 36                  LD      HL,RTCLOCK              ;Load the HL register with the address of the real time clock
0005  224813     37                  LD      (IntResp),HL            ;Load the interrupt response location with the address of the
                 38
0008  3E48       39                  LD      A,CTC0Interrupt         ;Load the A register with the CTC interupt vector.
000A  D380       40                  OUT     (CTC0INT),A             ;Write the interrupt vector to the CTC interupt register.
                 41
000C  3EB7       42                  LD      A,TimerMode             ;Load the A register with the CTC timer mode command.
000E  D380       43                  OUT     (CTC0CMD),A             ;Write the command to the CTC 0 command register.
                 44
0010  3E60       45                  LD      A,TimeConstant          ;Load the A register with the CTC time constant.
0012  D380       46                  OUT     (CTC0CMD),A             ;Write the time constant to the CTC command register.
                 47
0014  FB         48                  EI                              ;Enable system interrupts.
                 49
                 50
```

```
;       Equates for CLOCKINIT

CTC0CMD:                EQU             80H         ;Address of the CTC 0 command register.
CTC0INT:                EQU             80H         ;Address of the CTC 0 interrupt vector register.
TimerMode:              EQU             0B7H        ;CTC timer mode command.
CTC0Interrupt:          EQU             48H         ;CTC interupt vector for channel zero.
TimeConstant:           EQU             60H         ;Time constant for the CTC.
IntResp:                EQU             1346H       ;Address of the CTC0 interrupt response address.


;end: CLOCKINT=========================================================================


        ;begin: ATODINIT==========================================================================
        ;
        ;     Section ATODINIT initializes the analog to digital converter of the
```



```
LCC   OBJ CODE M STMT SOURCE STATEMENT
        ;AIO board into a polled mode.  This routine needs to be run after any
        ;system rest action.  ATODINIT takes {* 52.90 *} microseconds.
```

```
LOC   OBJ CODE M STMT SOURCE STATEMENT

                 68   ;AIO board into a polled mode.  This routine needs to be run after any
                 69   ;system rest action.  ATODINIT takes {* 52.90 *} microseconds.
                 70
                 71
0015  F3         72   ATODINIT:      DI                              ;Disable system interrupts.
                 73
0016  3E4F       74                  LD       A,InMode                         ;Load the A register with the PIO Mode 1 command.
0018  D322       75                  OUT      (CMD_A_PORT),A                   ;Write the command to the Port A command register.
001A  D323       76                  OUT      (CMD_B_PORT),A                   ;Write the command to the Port B command register.
                 77
001C  3E07       78                  LD       A,INTDisable                     ;Load the PIO interrupt disable command into A.
001E  D322       79                  OUT      (CMD_A_PORT),A                   ;Write the command to the Port A command register.
0020  D323       80                  OUT      (CMD_B_PORT),A                   ;Write the command to the Port B command register.
                 81
0022  DB20       82                  IN       A,(DataLower)                    ;Clear the lower data register.
0024  DB21       83                  IN       A,(DataUpper)                    ;Clear the upper data register.
                 84
0026  FB         85                  EI                                        ;Enable system interrupts.
                 86
                 87   ;       Equates for section ATCDINIT
                 88
                 89   InMode:                 EQU       4FH            ;PIO command for input mode.
                 90   INTDisable:             EQU       07H            ;PIO command to disable interrupts.
                 91   CMD_A_PORT:             EQU       22H            ;Address of PIO Port A command register.
                 92   CMD_B_PORT:             EQU       23H            ;Address of PIO Port B command register.
                 93   DataLower:              EQU       20H            ;Address of AtoD lower range data register.
                 94   DataUpper:              EQU       21H            ;Address of AtoD upper range data register.
                 95
                 96   ;end: ATODINIT===================================================================
                 97
                 98
```

```
                DataUpper:          EQU        21H          ;Address of AtoD upper range data register.

                ;end: ATODINIT===============================================================

0027  0C   LoopStart:          NOP


                ;begin: READATOD=============================================================
                ;
                ;       Section READATOD inputs the converted 12 bits of digital information
                ;from the analog to digital converter of the AIO board.  READATOD does this by
                ;reading the analog to digital converter status register.  Bit zero of the status
                ;word is a zero a conversion is inprogress.  If bit one of the status word is a
                ;zero a conversion is complete and has not been read.  Bit one is a one if the
                ;conversion has been read.  READATOD checks these two bits to see if a new conversion
                ;is available to the main routine.  If a new conversion is ready READATOD branches
                ;to DataIsReady to read in the new conversion value.  If a new (ie unread) conversion
                ;is not ready READATOD branches to READY? to check again.
                ;       The analog to digital channel being read from is determined by writing the
                ;desired channel number to the AtoDAtoDChannelSelect register.  READATOD
                ;assumes that this writing has been previously accomplished.

0028  DB29      READY?:        IN     A,(AtoDStatus)       ;Read the AtoD status register.
002A  CB47                     BIT    0,A                  ;Is the conversion complete ?
002C  28FA                     JR     Z,READY?             ;  No, try again.
                                                           ;  Yes, continue.
002E  DB20      DataIsReady:   IN     A,(DataLower)        ;Read the lower 8 bits.
0030  4F                       LD     C,A                  ;Load the lower bits into the C register.
0031  DB21                     IN     A,(DataUpper)        ;Read the upper 4 bits.
0033  47                       LD     B,A                  ;Load the upper bits into the B register.
```

```
                                TEST3        810322         PAGE    4
LOC   OBJ CODE M STMT SOURCE STATEMENT                       ASM 5.8

                ;       Equates for section READATOD

                AtoDStatus:    EQU              29H          ;Address of the status register.
                ;DataLower:    EQU              20H          ;Address of the lower data register.
                ;DataUpper:    EQU              21H          ;Address of the upper data register.

                ;end: READATOD===============================================================
```

```
                ;begin: OUTDA=====================================================================
                ;
                ;       OUTDA is a routine to output a 12 bit sign plus 2's complement representation
                ;to the channel one digital of analog converter on the AIO board.  The output word
                ;is passed to the routine in the BC register pair.  This routine does not alter any
                ;of the CPU registers.  OUTDA takes 15.25 microseconds to execute not including the
                ;subroutine call.  The maximum output settling time is 10 microseconds.  To address
                ;digital to analog converter channel two, substitute the address mnemonics DtoA2Lower
                ;and DtoA2Upper for DtoAlLower and DtoA2Upper in program lines 14 and 18.



0034   79       OUTDA:          LD      A,C                     ;Load the lower 8 bits of output into
                                                                ;       the A register.
0035   D32C                     OUT     (DtoAlLower),A          ;Write the lower 8 bits of output
                                                                ;       to the DtoA buffer.
0037   78                       LD      A,B                     ;Load the upper 4 bits of output into
                                                                ;       the A register.
0038   D32D                     OUT     (DtoAlUpper),A          ;Write the upper 4 bits of output
                                                                ;       to the DtoA buffer.

                ;       Equates for section CUTDA


                ;Use these mnemonics in lines 14 and 18 for DtoA channel one.

                DtoAlLower:     EQU     2CH             ;Address of DtoA channel one lower range.
                DtoAlUpper:     EQU     2DH             ;Address of DtoA channel one upper range.

                ;Use these mnemonics in lines 14 and 18 for DtoA channel two.

                DtoA2Lower:     EQU     2EH             ;Address of DtoA channel two lower range.
                DtoA2Upper:     EQU     2FH             ;Address of DtoA channel two upper range.

                ;end: OUTDA=======================================================================
```

```
                        ;begin: CHECKEND========================================================================
                        ;
                        ;       Routine CHECKEND reads the consol status word to determine if an escape
                        ;key has been depressed.  If so the current program is aborted and control of
                        ;the computer is returned to the RIO operating system.  The reading of the con-
                        ;sole status word is accomplished via the operating system through the call to
                        ;SYSTEM.  Routine CHECKEND is inteded to be a part of a larger program - it is
                        ;not written as a subroutine.

                                GLOBAL          CHECKEND

003A   FD215200 R       CHECKEND:       LD      IY,ConVector    ;Load the IY register with the console vector address.


                        TEST3           810322          PAGE    5
LOC    OBJ CODE M STMT SOURCE STATEMENT                  ASM 5.8

003E   CD0314                           CALL    SYSTEM          ;Call the operating system

0041   3A5C00   R                       LD      A,(CompCode)    ;Load the A register with the completion code for OK operation
0044   FE80                             CP      OKCode          ;Compair the OK code with the returned code OpCode.
0046   C20014                           JP      NZ,RIO          ;  An error has occured.  Jumpto the RIO reentry point.
                                                                ;  No error has occured.  Continue.
0049   3A5100   R                       LD      A,(ConStatus)   ;Load the A register with the console status word.

004C   CB6F                             BIT     5,A             ;Is an escape pending ?
004E   CA0014                           JP      Z,RIO           ;  Yes, jump to the RIO reeentry point.
                                                                ;  No escape is pending.  Continue program operation.
```

```
                                           ;  No escape is pending.  Continue program operation.


                         ;      Storage for routine CHECKEND

0051              ConStatus:    DEFS    1               ;Storage for the console status word.

0052  01                ConVector:    DEFB    ConIn           ;Storage for the logical unit code.
0053  40                              DEFB    RdRqstCode      ;Storage for the Request Code.
0054  5100    R                       DEFW    ConStatus       ;Storage for the address of the console status word.
0056  0100                            DEFW    1               ;Storage for the data length.
0058  0000                            DEFW    0               ;
005A  0000                            DEFW    0               ;
005C  00                CompCode:     DEFB    0               ;Storage for the address of Completion Code.
005D  0000                            DEFW    0               ;

                         ;      Equates for routine CHECKEND

                        RdRqstCode:   EQU     40H             ;Request code for Reading.
                        ConIn:        EQU     01H             ;Logical unit code for concole input.
                        OKCode:       EQU     80H             ;Completion code for successful operation.
                        SYSTEM:       EQU     1403H           ;Address of system routine entry point.
                        RIO:          EQU     1400H           ;Address of RIO operating system reentry point.

                        ;end: CHECKEND===============================================================


005F  C32700  R                 JP      LoopStart             ;Do the TEST3 loop again.


                  ;end: TEST3=====================================================================


```

```
                    226
                    227
                    228
                    229 ;begin: RTCLOCK=====================================================================================
                    230 ;
                    231 ;       Section RTCLOCK is the interrupt service routine to manage the real time
                    232 ;clock function for TOTAL.  RTCLOCK is called each time the CTC timer reaches zero.
                    233 ;When this occurs RTCLOCK resets the CTC timer and initiates an analog to digital
                    234 ;conversion on the desired channel, indicated by the constant Channel.
                    235
                    236         GLOBAL          RTCLOCK
                    237
0062   3E00         238 RTCLOCK:        LD      A,Channel               ;Load the A register with the desired A/D channel number.
0064   D328         239                 OUT     (AtoDChannelSelect),A   ;Write the channel number to the channel select register.
                    240
0066   FB           241                 EI                              ;Enable interrupts.

                        TEST3           810322              PAGE   6
 LOC   OBJ CODE M STMT SOURCE STATEMENT                     ASM 5.8

                    242
0067   ED4D         243                 RETI                            ;Return from the interrupt to the original routine.
                    244
                    245
                    246 ;       Equates for section RTCLOCK
                    247
                    248 Channel:                EQU             00H             ;Channel zero of the analog to digital converter.
                    249 AtoDChannelSelect:      EQU             28H             ;The address of the AtoD channel select register.
                    250
                    251 ;end: RTCLOCK=======================================================================================
                    252
                    253
                    254         END
```

```
SYMBOL  VAL M DEFN REFS

ATODIN 0015 G   72    6
AtoDCh 0028    249  239
AtoDSt 0029    128  117
CHECKE 003A G  183    6  181
CLOCKI 0001 G   34   32
CMD_A_ 0022     91   75   79
CMD_B_ 0023     92   76   80
CTC0CM 0080     54   43   46
CTC0IN 0080     55   40
CTC0In 0048     57   39
Channe 0000    248  238
CompCo 005C R  208  187
ConIn  0001    214  202
ConSta 0051 P  200  191  204
ConVec 0052 P  202  183
DataIs 002E G  121    6
DataLo 0020     93   82  121
DataUp 0021     94   83  123
DtoA1L 002C    162  150
DtoA1U 002D    163  154
DtoA2L 002E    167
DtoA2U 002F    168
INTDis 0007     90   78
InMode 004F     89   74
IntRes 1348     59   37
LoopSt 0027 G   99    6  222
OKCode 0080    215  188
OUTDA  0034 G  148    6
READY? 0028 G  117    6  119
RIO    1400    217  189  194
RTCLOC 0062 G  238   36  236
RdRqst 0040    213  203
SYSTEM 1403    216  185
TEST3  0000 G    3    6
TimeCo 0060     58   45
TimerM 00B7     56   42
```

Appendix E: Buffers Module Listings

The following page is the compiler listing of the Buffers Module.

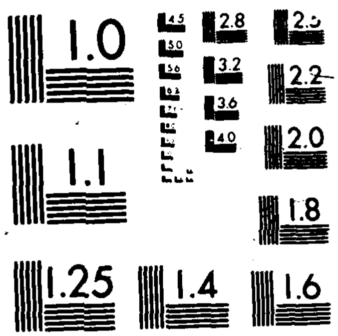
```
PLZSYS 3.0  810303.1216
       BUFFERS MODULE                                      ! 3 March 1981 !

       !        This module is part of the data collection system program.  It is  !
       ! the last of the modules in memory and establishes the beginning address of  !
       ! the data buffer into which the data from the A to D converters is written.  !

        CONSTANT

          BUFFER_SIZE := 1000



        TYPE

          BUFFER  ARRAY [ BUFFER_SIZE  WORD ]



        GLOBAL

          DATA_BUFFER   BUFFER

        -

        END BUFFERS

END OF COMPILATION:    0 ERROR(S)      0 WARNING(S)
 2000 DATA BYTES       0 Z-CODE BYTES    SYMBOL TABLE  1% FULL
```

Appendix F: Collect_Data Module Listings

The following pages are a listing of the Collect_Data Module. This is not a compiled or assembled listing; it is source code.

AD-A172 823 DESIGN AND PARTIAL IMPLEMENTATION OF A COMPUTER CONTROLLED DATA COLLECTION SYSTEM(U) AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH SCHOOL OF ENGI.. L E LUTZ 5/5

UNCLASSIFIED FEB 86 AFIT/GE/ENG/86M-1 F/G 9/2 NL


1.0
2.8
2.5
3.2
2.2
3.6
4.0
2.0
1.1
1.8
1.25
1.4
1.6

```
COLLECT_DATA MODULE                                      ! 3 March 1981 !

    !     This module is the COLLECT_DATA block of the data collection system !
    ! structure diagram.  There are five subelements for this block.  They    !
    ! are:                                                                    !
    !         1. CREATE_DATA_FILE                                             !
    !         2. PREPARE_COLLECTOR                                            !
    !         3. SAMPLE_DATA                                                  !
    !         4. LOAD_DATA_FILE                                               !
    !         5. CLOSE_DATA_FILE                                              !
    ! each of which is a procedure called by SAMPLE_DATA.                     !
    !                                                                         !
```

```
CONSTANT      !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

   MICRO_SECONDS := -6
   MILLI_SECONDS := -3
   SECONDS       := 0
   FAST_MODE     := %87          ! CTC command for interrupting timer with prescale of 16, time constant to follow.  !
   SLOW_MODE     := %A7          ! CTC command for interrupting timer with prescale of 256, time constant to follow. !
   END_OF_STRING := '|'          ! Character used to define the end of a string of ASCII characters.               !
   END_OF_FILE   := %FF          ! RIO end of file designator.                                                       !
   CARRIAGE_RETURN := %0D        ! ASCII carriage return.  PLZ also uses %R for carriage return.                    !

   MINIMUM_TIME := 50            ! The shortest time in microseconds the interrupt timer can be programmed for.     !

   CONSOLE_OUT  := 2             ! The logical unit number for the system CONOUT, console output device.            !
   DATA_FILE    := 7             ! The logical unit number assigned to the disk file for raw data storage.          !

   BUFFER_SIZE        := %1000   ! Since the memory manager cannot be directly called from a PLZ program, these two !
   MAX_BUFFER_ADDRESS := %9A00   ! constants are used to form the raw data buffer.                                  !

       ! ERROR_CODE constants !
   FALSE              := %00     ! No error has occured, all_is_ok.                                                 !
   FATAL              := %FF     ! The error is fatal, hault operation of program.                                  !
   ABORT              := %AD     ! The user has aborted the program operation.                                      !
   TOO_MANY_SAMPLES   := %E0     ! The number of samples requested by the user exceeds available memory space.      !
   BAD_CHARACTER      := %BC     ! One of the characters in a file name string ( or protion of ) is invalid.        !
   PERIOD_RANGE_ERROR := %E1     ! The timing range specified by the user is outside the defined bounds.            !
   REDO               := %22     ! There has been an error in user input, repeat the input process.                 !
   STORAGE_ERROR      := %23     ! There was a system error during the copy of raw data from memory to disk.        !

       ! Operating System Return Codes !
   OPERATION_COMPLETE  := %80    ! Return code for successful operation.                                             !
   DUPLICATE_FILE      := %D0    ! File of same name already exists.                                                 !
   INSUFFICIENT_MEMORY := [illegible] ! Insufficient memory is available.                                            !
   DEVICE_NOT_READY    := %C2    ! Requested device not available at this time.                                      !
   FILE_NOT_FOUND      := %C7    ! Requested file not found on either system disk.                                   !
```

```
TYPE          !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

   PBYTE ^BYTE
   ASCII_STRING  ARRAY[ 32 BYTE ]
   ASCII_PTR     ^ASCII_STRING

   BUFFER        ARRAY[ BUFFER_SIZE  WORD ]       ! Array for raw data storage. !
   BUFFER_POINT  ^BUFFER                          ! A pointer to a BUFFER.      !
```

```
EXTERNAL     !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

   SAMPLER PROCEDURE ( CTC_MODE CTC_TIME_CONSTANT CHANNEL  BYTE,
                       COUNT SAMPLES  WORD,
                       FIRST_DATA_LOCATION  PBYTE )
           RETURNS   ( ERROR_CODE  BYTE, LAST_DATA_LOCATION  PBYTE )

   DATE PROCEDURE
        RETURNS   ( YEAR1 YEAR0 MONTH1 MONTH0 DAY1 DAY0  BYTE )

   WRITE_HBYTE PROCEDURE ( LOGICAL_UNIT VALUE  BYTE )

   WRITE_HINTEGER PROCEDURE ( LOGICAL_UNIT  BYTE, VALUE  INTEGER )

   WRITE_HWORD PROCEDURE ( LOGICAL_UNIT  BYTE, VALUE  WORD )

   WRITE_DWORD PROCEDURE ( LOGICAL_UNIT  BYTE, VALUE  WORD )

   WRITE_RCODE PROCEDURE ( LOGICAL_UNIT RETURN_CODE  BYTE )

   WRITELN_RCODE PROCEDURE ( LOGICAL_UNIT RETURN_CODE  BYTE )

   WRITE   PROCEDURE ( LOGICAL_UNIT  BYTE, TEXT_POINTER PBYTE )

   WRITELN PROCEDURE ( LOGICAL_UNIT  BYTE, TEXT_POINTER PBYTE )

   OPEN    PROCEDURE ( LOGICAL_UNIT BYTE, FILE_NAME_PTR PBYTE, MODE BYTE )
           RETURNS   ( RETURN_CODE  BYTE )

   CLOSE   PROCEDURE ( LOGICAL_UNIT BYTE )
           RETURNS   ( RETURN_CODE  BYTE )

   SEEK    PROCEDURE ( LOGICAL_UNIT BYTE, POSITION_HIGH POSITION_LOW WORD, MODE BYTE )
           RETURNS   ( RETURN_CODE  BYTE )

   PUTSEQ  PROCEDURE ( LOGICAL_UNIT BYTE, BUFFER_PTR PBYTE, NUMBER_OF_BYTES WORD )
           RETURNS   ( RETURN_CODE BYTE )

   DATA_BUFFER BUFFER         ! The raw data buffer is in the last module of the program. !
```

```
INTERNAL    !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

  STRING_COPY PROCEDURE ( SOURCE ASCII_PTR, S_INDEX BYTE, DESTINATION ASCII_PTR, D_INDEX BYTE )

  !     This routine copies the string pointed to by SOURCE ( beginning at !
  ! location S_INDEX - ending at the location containing an end of string  !
  ! character '|' ) onto the string pointed to by DESTINATION ( beginning  !
  ! at location D_INDEX -  ending when SOURCE's '|' is read.  DESTINATION  !
  ! is returned with a '|' in the last position.                           !

   ENTRY
     DO
       IF SOURCE^[ S_INDEX ] = END_OF_STRING THEN EXIT FI
       DESTINATION^[ D_INDEX ] := SOURCE^[ S_INDEX ]
       S_INDEX := S_INDEX + 1
       D_INDEX := D_INDEX + 1
     OD
     DESTINATION^[ D_INDEX ] := END_OF_STRING
   END STRING_COPY


  ASCII PROCEDURE ( NUMBER INDEX DIVISOR WORD, INPOINTER ASCII_PTR )
        RETURNS   ( TEXT_STRING ASCII_PTR )

   LOCAL
     VALUE WORD
     POINT INTEGER
   ENTRY
     TEXT_STRING^[0] := INPOINTER^[0]
     POINT := 0
     DO
       VALUE := NUMBER / INDEX
       NUMBER := NUMBER MOD INDEX
       TEXT_STRING^[ POINT ] := '?'
       IF VALUE
         CASE %0 THEN  TEXT_STRING^[ POINT ] := '0'
         CASE %1 THEN  TEXT_STRING^[ POINT ] := '1'
         CASE %2 THEN  TEXT_STRING^[ POINT ] := '2'
         CASE %3 THEN  TEXT_STRING^[ POINT ] := '3'
         CASE %4 THEN  TEXT_STRING^[ POINT ] := '4'
         CASE %5 THEN  TEXT_STRING^[ POINT ] := '5'
         CASE %6 THEN  TEXT_STRING^[ POINT ] := '6'
         CASE %7 THEN  TEXT_STRING^[ POINT ] := '7'
```

```
  TODAYS_DATE^[2] := MONTH1
  TODAYS_DATE^[3] := MONTH2
  TODAYS_DATE^[4] := DAY1
  TODAYS_DATE^[5] := DAY2
  TODAYS_DATE^[6] := CARRIAGE_RETURN
END GET_DATE
```

```
FIND_CTC_COMMANDS PROCEDURE ( TIME UNITS  INTEGER )
                  RETURNS   ( ERROR_CODE CTC_MODE CTC_TIME_CONSTANT  BYTE,  COUNT  WORD )

!     FIND_CTC_COMMANDS takes the user input timing period in TIME and  !
! UNITS and determines the proper routine and parameters for that routine !
! for the interrupt service routine to produce the user's desired period. !
! UNITS is the order-of-magnitude of the timeing period.  It can take on  !
! the values MICRO_SECONDS, MILLI_SECONDS, or SECONDS, which are all con- !
! stants.  TIME is the number of UNITS that the user desires the timing   !
! period to be.  This routine determines wether the timer routine alone   !
! is sufficient for the period ( period less than 100 milliseconds ) or   !
! whether the counter-timer routine is needed.  If no counter is needed,  !
! COUNT is returned with a value of zero.  There are two possible timing  !
! modes for the timer, fast and slow.  In the fast mode the CTC prescale  !
! factor is 16, in the slow mode it is 256.  The constants FAST_MODE and  !
! SLOW_MODE are the CTC commands for interrupting timer with the appro-   !
! piate prescaling factor.  FIND_CTC_COMMANDS determines which mode is to !
! be used.  Lastly the routine, when the counter is needed, finds the     !
! down-counter value necessary.  When the counter is used the timer is    !
! set to either .001 seconds or .025 seconds.  The value of COUNT is then !
! determined to yield the user's timing period.                           !
!     This routine calls procedure FIND_TIME_CNST to determine the best   !
! ( least error ) value for CTC_TIME_CONSTANT.                            !

  ENTRY
    IF UNITS

      CASE MICRO_SECONDS THEN                         ! Timing periods of [illegible] microseconds.  !
        IF ( TIME < MINIMUM_[illegible] )             ! If the timing period is less than the        !
          THEN                                        ! minimum allowed value.                       !
            ERROR_CODE := PERIOD_RANGE_ERROR
          ELSE                          ! Timer only interrupt service routine.  Use a CTC prescale factor of !
            COUNT := 0                  ! 16.  Break the time constant determination down into the most accur !
            CTC_MODE := FAST_MODE       ! ate ranges.  This action is required due to truncating division.    !
            IF TIME <= 26
              THEN CTC_TIME_CONSTANT := FIND_TIME_CNST( WORD( TIME ), 2457, 16000 )
              ELSE IF TIME <= 266
                THEN CTC_TIME_CONSTANT := FIND_TIME_CNST( WORD( TIME ), 246, 1600 )
                ELSE CTC_TIME_CONSTANT := FIND_TIME_CNST( WORD( TIME ),  25,  160 )
                FI
```

```
          FI
        FI

      CASE MILLI_SECONDS THEN
        IF TIME < 0 ORIF TIME > 999                          ! If the time is out of range.          !
          THEN ERROR_CODE := PERIOD_RANGE_ERROR
          ELSE
            IF TIME < 10                                   ! Determine which routine is needed.       !

              THEN                                           ! Timer only routine is needed.          !
                COUNT := 0                                       ! Zero the down counter value.       !
                CTC_MODE := SLOW_MODE                            ! CTC prescale factor of 256.        !
                CTC_TIME_CONSTANT := FIND_TIME_CNST( WORD( TIME ), 2457, 256 )

              ELSE                                           ! Counter-timer routine is needed.       !
                CTC_MODE := FAST_MODE                            ! CTC prescale factor of 16.             !
                CTC_TIME_CONSTANT := 154                         ! Set the timer for a 1 millisecond period. !
                COUNT := WORD( TIME )                                    ! Set the down-counter value to the proper  !
                                                                 ! value, number of 1 millisecond periods.   !
              FI
          FI

      CASE SECONDS THEN
        CTC_MODE := SLOW_MODE                                ! Set timer for 25 milliseconds.         !
        CTC_TIME_CONSTANT := 240
        COUNT := WORD( TIME ) * 40
                                                                   ! Set the down-counter value to the proper  !
                                                             ! value, 40 * 25 milliseconds = seconds,    !
                                                             ! number of 25 millisecond periods.         !

      FI                                                   ! End of CASEs of UNITS.                !

    END FIND_CTC_COMMANDS
```

```
SIZE_DATA_BUFFER PROCEDURE ( SAMPLES_REQUESTED  WORD )
                 RETURNS   ( ERROR_CODE  BYTE, SAMPLES_ALLOWED  WORD )

!    This procedure crudely performs a memory manager function in an    !
! open loop mode.  Via the constants BUFFER_SIZE and MAX_BUFFER_ADDRESS, !
! the external buffer DATA_BUFFER, and the parameter SAMPLES_REQUESTED  !
! SIZE_DATA_BUFFER determines whether there is sufficient memory avail- !
! able for the number of samples desired.  If not the number of samples !
! is reduced to the maximum possible.  The user must set MAX_BUFFER_ADDR !
! ESS to the address of the next highest free byte in memory above the  !
! linking of this program.  The function of this routine could be made  !
! closed loop by calling the system memory manager via an assembly lan- !
! guage routine.                                                         !

 LOCAL
   AVAILABLE_WORDS  WORD

  ENTRY
    AVAILABLE_WORDS := ( MAX_BUFFER_ADDRESS - #DATA_BUFFER[0] ) / 2  ! Find the number of WORDS there is space for. !
    IF AVAILABLE_WORDS < SAMPLES_REQUESTED
      THEN                                    ! The request cannot be honored.  Fewer samples will be allowed. !
        ERROR_CODE := TOO_MANY_SAMPLES
        SAMPLES_ALLOWED := AVAILABLE_WORDS
      ELSE                                    ! The request can be honored. !
        SAMPLES_ALLOWED := SAMPLES_REQUESTED
        ERROR_CODE  := FALSE
      FI
 END SIZE_DATA_BUFFER
```

```
ERROR_IN_PREPARE  PROCEDURE ( IN_ERROR_CODE   BYTE )
                  RETURNS   ( OUT_ERROR_CODE  BYTE )

  ENTRY
     IF ERROR_CODE
       CASE TOO_MANY_SAMPLES THEN
         WRITE( CONSOLE_OUT, #' Too many samples requested, %R' )
         WRITE_DWORD( CONSOLE_OUT, NUM_SAMPLES )
         WRITELN( CONSOLE_OUT, #' will be collected.%R' )
         ERROR_CODE := FALSE
       CASE PERIOD_RANGE_ERROR THEN
         WRITELN( CONSOLE_OUT, #' Sampling period specified is outside of defined ranges.%R' )
         WRITELN( CONSOLE_OUT, #'Defined ranges:   7 to 999 microseconds.%R' )
         WRITELN( CONSOLE_OUT, #'                  1 to 999 milliseconds.%R' )
         WRITELN( CONSOLE_OUT, #'                  1 to 1700 seconds.%R' )
         WRITELN( CONSOLE_OUT, #'Please use a valid sampling period.%R' )
         ERROR_CODE := FATAL
      FI
 END ERROR_IN_PREPARE
```

```
PREPARE_COLLECTOR  PROCEDURE ( PERIOD_VALUE PERIOD_UNITS  INTEGER,
                               SAMPLES_REQUESTED  WORD )
                   RETURNS   ( ERROR_CODE CTC_MODE TIME_CONSTANT  BYTE,
                               DOWN_COUNT NUMBER_OF_SAMPLES  WORD )

  ENTRY
    DO

      ERROR_CODE, CTC_MODE, TIME_CONSTANT, DOWN_COUNT := FIND_CTC_COMMANDS( PERIOD_VALUE PERIOD_UNITS )
      IF ERROR_CODE <> FALSE THEN
        ERROR_CODE := ERROR_IN_PREPARE( ERROR_CODE )
        EXIT
        FI

      ERROR_CODE, NUMBER_OF_SAMPLES := SIZE_DATA_BUFFER( SAMPLES_REQUESTED )
      IF ERROR_CODE <> FALSE THEN
        ERROR_CODE := ERROR_IN_PREPARE( ERROR_CODE )
        EXIT
        FI

      EXIT
    OD
END PREPARE_COLLECTOR
```

```
!!!!!!!!!!! create_data_file routines !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!


ERROR_IN_CREATE PROCEDURE ( IN_ERROR_CODE RETURN_CODE  BYTE )
                RETURNS   ( OUT_ERROR_CODE BYTE )

 LOCAL

 ENTRY
   IF IN_ERROR_CODE = BAD_CHARACTER
     THEN
       WRITELN( CONSOLE_OUT_OUT, #'Invalid character in testid detected by VALID_STRING in COLLECT_DATA.%R' )
       WRITELN( CONSOLE_OUT_OUT, #'Fatal error.  Program terminated.%R' )
       OUT_ERROR_CODE := FATAL
     ELSE
       IF RETURN_CODE := DUPLICATE_FILE
         THEN
           WRITELN( CONSOLE_OUT, #'Raw data file with same TESTID already exists.%R' )
           WRITELN( CONSOLE_OUT, #'Use a new TESTID.%R' )
           OUT_ERROR_CODE := FATAL
         ELSE
           WRITE_RCODE( CONSOLE_OUT, RETURN_CODE )
           WRITELN( CONSOLE_OUT, #', return_code from open data file request.%R' )
           OUT_ERROR_CODE := FATAL
         FI
       FI
   END ERROR_IN_CREATE
```

```
VALID_STRING PROCEDURE ( TEST_STRING  ASCII_PTR )
             RETURNS   ( ERROR_CODE  BYTE )

!     This routine examines the contents of the string pointed to by  !
! TEST_STRING.  If the string contains any characters not allowed in  !
! a file name then ERROR_CODE is returned with a value of FATAL.  If  !
! all characters are valid ERROR_CODE is returned with a value of     !
! FALSE.  The check continues until an END_OF_STRING is seen or unitl !
! 32 characters have been examined.                                    !

  LOCAL
    INDEX  BYTE
  ENTRY
    INDEX := 0
    ERROR_CODE := FALSE
    DO
      IF ( TEST_STRING^[ INDEX ] < %30 ) ORIF     ! If below '0' !
         ( ( TEST_STRING^[ INDEX ] >%3A ) ANDIF ( TEST_STRING^[ INDEX ] < %41 ) ) ORIF ! If character is punctuation !
         ( TEST_STRING^[ INDEX ] > %5A )          ! If character is above 'Z' !
        THEN ERROR_CODE := BAD_CHARACTER FI
      INDEX := INDEX + 1
      IF ( TEST_STRING^[ INDEX ] = END_OF_STRING ) ORIF
         ( INDEX >= 32 )                           ORIF
         ( ERROR_CODE <> FALSE )
        THEN EXIT FI
    OD
  END VALID_STRING
```

```
CREATE_DATA_FILE PROCEDURE ( INPUT_CHANNEL DATA_FILE   BYTE,
                             PERIOD_VALUE PERIOD_UNITS  INTEGER,
                             SAMPLES   WORD,
                             TESTID USER_MESSAGE TODAYS_DATE  ASCII_PTR )
                 RETURNS   ( ERROR_CODE  BYTE )

 LOCAL
   FILE_NAME_BUF  CHANNEL_BUF  ASCII_STRING
   FILE_NAME      CHANNEL      ASCII_PTR
   RETURN_CODE    BYTE

 ENTRY
   FILE_NAME    := #FILE_NAME_BUF[0]
   CHANNEL      := #CHANNEL_BUF[0]

   COPY_STRING( TESTID, 0, FILE_NAME, 0 )
   FILE_NAME^[6] := '.'

   CHANNEL := ASCII( WORD( INPUT_CHANNEL), 10, 10, CHANNEL )
   COPY_STRING( CHANNEL, 0, FILE_NAME, 7 )

   FILE_NAME^[9] := '.'
   COPY_STRING( #'RAW_DATA$R|', 0, FILE_NAME, 10 )

   RETURN_CODE := OPEN( DATA_FILE, FILE_NAME, CREATE_MODE )

   IF RETURN_CODE <> OPERATION_COMPLETE
     THEN
       ERROR_CODE := FATAL              ! A system error has occured, diagnosis and inform the user. !
       ERROR_CODE := ERROR_IN_CREATE( ERROR_CODE, RETURN_CODE )

     ELSE  ! No errors have occured, continue operation of create_data_file. !
       WRITE( DATA_FILE, #'testid:$R' )
```

```
    ELSE  ! No errors have occured, continue operation of create_data_file. !
      WRITE( DATA_FILE, #'testid:%R' )
      WRITE( DATA_FILE, TESTID )

      WRITE( DATA_FILE, #'|input_channel:%R' )
      WRITE_HBYTE( DATA_FILE, CHANNEL )

      WRITE( DATA_FILE, #'|period_value:%R' )
      WRITE_HINTEGER( DATA_FILE, PERIOD_VALUE )

      WRITE( DATA_FILE, #'|period_units:%R' )
      WRITE_HINTEGER( DATA_FILE, PERIOD_UNITS )

      WRITE( DATA_FILE, #'|#_samples:%R' )
      WRITE_HWORD( DATA_FILE, SAMPLES )

      WRITE( DATA_FILE, #'|date_of_test:%R' )
      WRITE( DATA_FILE, TODAYS_DATE )

      WRITE( DATA_FILE, #'|user_message:%R' )
      WRITE( DATA_FILE, USER_STRING )

      WRITELN( DATA_FILE, #'|beginning of data:||%R' )
    FI

END CREATE_DATA_FILE
```

```
!!!!!!!!!! load_data_file routines !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

LOAD_DATA_FILE  PROCEDURE ( DATA_FILE  BYTE,
                            BUFFER_BEGINNING LAST_DATA  PBYTE )
                RETURNS   ( ERROR_CODE )

 LOCAL
   NUMBER_OF_BYTES  BYTES_WRITTEN  WORD
   RETURN_CODE    BYTE
 ENTRY
   NUMBER_OF_BYTES := ( LAST_DATA - BUFFER_BEGINNING ) + 1
   BYTES_WRITTEN, RETURN_CODE := PUTSEQ( DATA_FILE BUFFER_BEGINNING NUMBER_OF_BYTES )
   IF BYTES_WRITTEN <> NUMBER_OF_BYTES THEN
     ERROR_CODE := STORAGE_ERROR
     WRITE( CONSOLE_OUT, #'Error in data transfer to disk.  %R' )
     WRITE_DWORD( CONSOLE_OUT, NUMBER_OF_BYTES )
     WRITELN( CONSOLE_OUT, #'  stored in memory,%R' )
     WRITE_DWORD( CONSOLE_OUT, BYTES_WRITTEN )
     WRITELN( CONSOLE_OUT, #' transfered to disk.%R' )
     FI

   IF RETURN_CODE <> OPERATION_COMPLETE THEN
     ERROR_CODE := FATAL
     WRITELN( CONSOLE_OUT, #'Fatal system error upon data transfer to disk.%R' )
     WRITE( CONSOLE_OUT, #'    System return code: %R' )
     WRITELN_BCODE( CONSOLE_OUT, RETURN_CODE )
     FI
 END LOAD_DATA_FILE
```

```
!!!!!!!!!! close_data_file routines !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

CLOSE_DATA_FILE PROCEDURE ( DATA_FILE  BYTE )
                RETURNS   ( ERROR_CODE      BYTE )

  LOCAL
    RETURN_CODE BYTE
  ENTRY
    RETURN_CODE := CLOSE( DATA_FILE )
    IF RETURN_CODE <> OPERATION_COMPLETE THEN
      WRITELN( CONSOLE_OUT, #'System fatal error upon data file close request.%R')
      WRITE( CONSOLE_OUT, #'     System return code: %R' )
      WRITELN( CONSOLE_OUT, RETURN_CODE )
      ERROR_CODE := FATAL
      FI
  END CLOSE_DATA_FILE
```

```
!!!!!!!!!!! sampler routines !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

  ERROR_IN_SAMPLER PROCEDURE ( IN_ERROR_CODE   BYTE )
                   RETURNS   ( OUT_ERROR_CODE  BYTE )

    ENTRY
      IF IN_ERROR_CODE = ABORT
        THEN  WRITELN( CONSOLE_OUT, $' User abort of program detected by routine SAMPLER.$R' )
        ELSE  WRITELN( CONSOLE_OUT, $' Fatal error occured in routine SAMPLER, program is terminated.$R' )
        FI
      WRITELN( CONSOLE_OUT, $' All files will be closed but not deleted.$R' )
      ERROR_CODE := CLOSE_DATA_FILE( FILE_UNIT )
      ERROR_CODE := FATAL
    END ERROR_IN_SAMPLER
```

```
GLOBAL      !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

  SAMPLE_DATA  PROCEDURE  ( TESTID USER_MESSAGE  ASCII_STRING,
                            PERIOD_VALUE PERIOD_UNITS  INTEGER,
                            INPUT_CHANNEL  BYTE,
                             SAMPLES  WORD )
               RETURNS    ( ERROR_CODE  BYTE )

    LOCAL
      TODAYS_DATE_BUF  ASCII_STRING
      TODAYS_DATE      ASCII_PTR
      LAST_DATA        PBYTE

    ENTRY
      ERROR_CODE := FALSE
      TODAYS_DATE := ^TODAYS_DATE_BUF[0]
      TODAYS_DATE := GET_DATE( TODAYS_DATE )
      DO

        ERROR_CODE, CTC_MODE, TIME_CNST, DOWN_COUNT, NUM_SAMPLES :=
          PREPARE_COLLECTOR( PERIOD_VALUE PERIOD_UNITS SAMPLES )
        IF ERROR_CODE <> FALSE THEN EXIT FI

        ERROR_CODE :=
          CREATE_DATA_FILE( TESTID INPUT_CHANNEL FILE_UNIT PERIOD_VALUE PERIOD_UNITS SAMPLES USER_MESSAGE TODAYS_DATE )
        IF ERROR_CODE <> FALSE THEN EXIT FI

        ERROR_CODE, LAST_DATA :=
          SAMPLER( INPUT_CHANNEL CTC_MODE TIME_CNST DOWN_COUNT NUM_SAMPLES BEGINNING_OF_BUFFER )
        IF ERROR_CODE <> FALSE THEN ERROR_CODE := ERROR_IN_SAMPLER( ERROR_CODE  FILE_UNIT ) FI
        IF ERROR_CODE <> FALSE THEN EXIT FI

        ERROR_CODE := LOAD_DATA_FILE( DATA_FILE BEGINNING_OF_BUFFER LAST_DATA )
        IF ERROR_CODE <> FALSE THEN EXIT FI

        ERROR_CODE := CLOSE_DATA_FILE( DATA_FILE )
        EXIT

      OD

  END SAMPLE_DATA

END COLLECT_DATA
```

Appendix G: AIO.PLZ.S Module Listings

Introduction to AIO.PLZ.S Module

To determine how to use the AIO Analog Input Output board of the MCB Z-8Ø development system, the PLZ language routines of the AIO.PLZ.S Module were written. These routines permitted the initial operation and checkout of the board and served as software "breadboards" for the assembly language routines of Sampler Module actually employed in the final software.

The five PLZ language routines of AIO.PLZ.S Module and their functions are:

AIO_INIT: Initializes the AIO board;

IN_CHAN_SEL: Selects one of the sixteen analog-to-digital input channels and initiates the conversion;

IN_DIGITALP: Reads in data from the selected input channel;

I
N_DIGITALT: Selects an input channel, initiates analog-to-digital conversion, and reads data in from the channel; and

OUT_ANALOG: Outputs data on a selected digital-to-analog channel.

To accomplish these functions, these five PLZ routines use four external assembly language procedures from the Utilities Module. The routines and their functions are:

IOOUT: Writes a byte to an input/output port,
IOIN: Reads a byte from an input/output port,
ENABLEINT: Enables the CPU interrupts, and
DISABLEINT: Disables the CPU interrupts.

The relationship between the AIO.PLZ.S Module routines, calling routines, and the Utility Modue routines is shown in Figure 76 below.

Three of the AIO.PLZ.S routines, AIO_INIT, IN_CHAN_SEL, and IN_ DIGITALP, were initially used in in this thesis effort. They were replaced with assembly language versions of these routines to yield greater speed of execution. The AIO.PLZ.S Module routines obtain access to the AIO board through two other modules, UTILITY and PLZ STREAM.IO. The assembly language routines directly communicate with the AIO board. The PLZ routines however, were quite helpful during initial development of the higher level modules of the thesis effort.

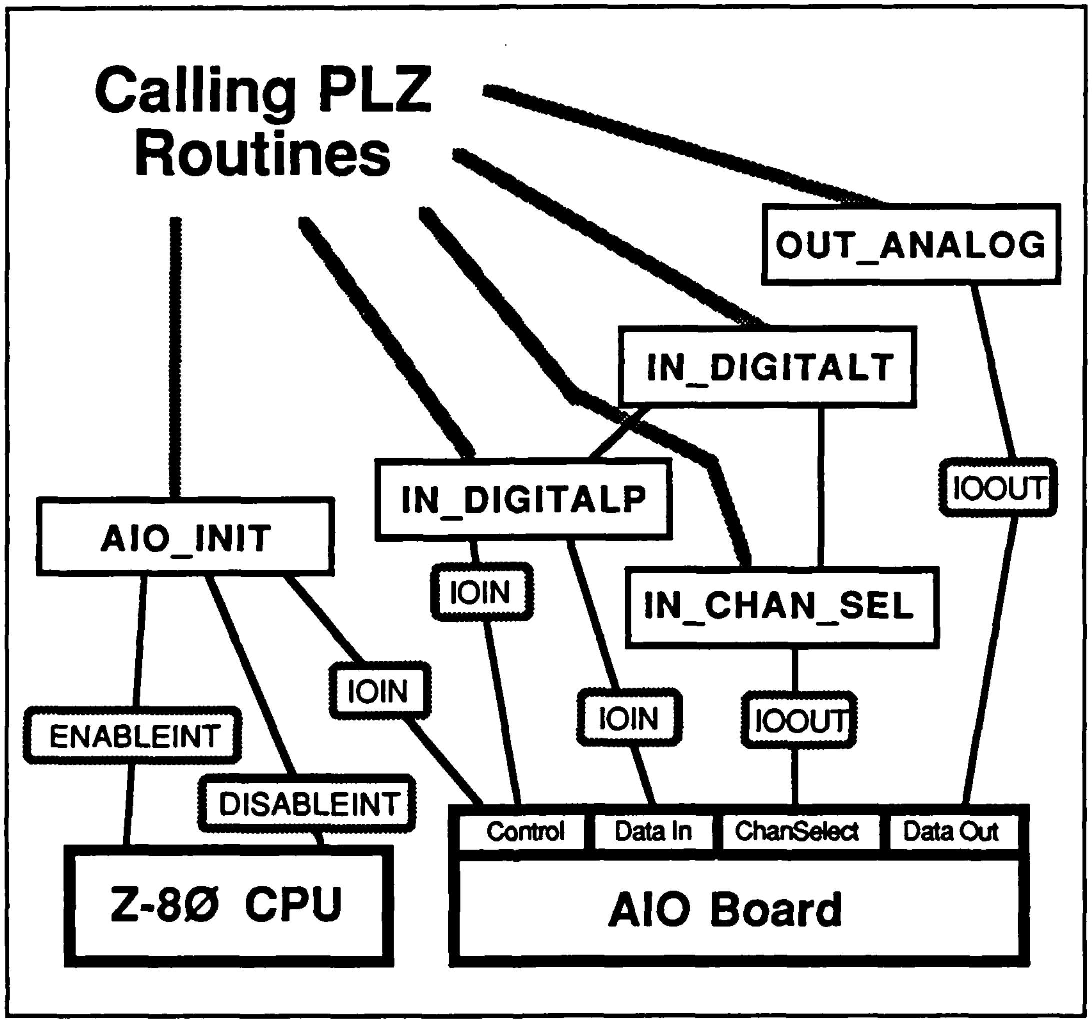


Figure 76. Relationship of AIO.PLZ.S Routines to Their Calling Routines, the Routines of the Utility Module, and to System Elements.

The following pages detail the five PLZ language routines of the AIO. PLZ.S Module. For each routine the following information will be presented.

1. The name of the routine.
2. The name of the routine's module.
3. The language of the routine and the number of lines of code.
4. A synopsis of the routine.
5. A diagram showing the relationship of the routine with other routines, both calling and called.
6. How the routine is invoked including parameter passing schema and a list of the calling routines.
7. A list and description of the global, module, and routine level variables and constants.
8. A list of the other routines called including a description of their function and their parameter passing schema.
9. Descriptions of the output parameters of the routine and any system configuration changes it makes.
10. A discussion of the test performed on the routine and the results of those tests.
11. A reference to the program listing of the routine.

1. Routine Name: **AIO_INIT**

2. Part of AIO.PLZ.S Module

3. Written in PLZ; nine lines of executable code.

4. Synopsis of Routine

AIO_INIT initializes the AIO Analog Input Output board of the Z-8Ø development system. To prevent inadvertent interrupts during this initialization process, the first action of AIO_INIT is to call the external routine DISABLEINIT. The AIO initialization is accom- plished by writing commands to the control ports of the board. The external routine IOOUT is used for this writing. The AIO board is put into polled mode and inhibited from issueing interupts.. The input registers of the AIO board are then cleared by reading them via the external routine IOIN. Lastly, the system interrupts are enabled by calling the external routine ENABLEINT.

5. Routine Relationships Diagram

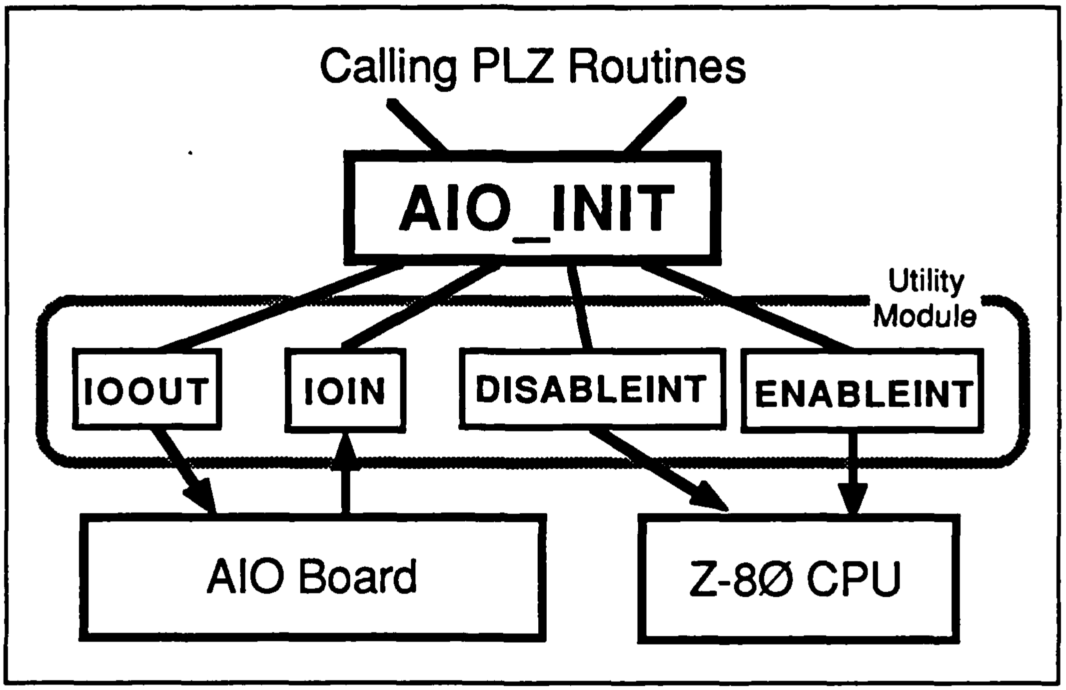


Figure 77. Relationship of AIO_INIT to Calling PLZ Routine and the External Routines.

6. Invocation

a. Invocation Statement

AIO_INIT is invoked solely by its name. To be invoked however, both the AIO.PLZ.S and UTILITY modules must be linked in with the calling routine's module.

b. Parameter Passing Schema

There are no input parameters for AIO_INIT.

c. Routines Which Call

AIO_INIT can be called by any PLZ routine using the AIO board. For this thesis effort, AIO_INIT was used during initial work with the AIO board. For the combined modules of the thesis effort, and assembly language program, AIOINIT, similar in function to AIO_INIT, was used.

7. Variables and Constants

a. Global

AIO_INIT uses no globally defined variables or constants.

b. Module

AIO_INIT uses six module constants for AIO board addresses and commands. The six are:

COMMAND_UPPER: value 23h, address of upper AIO command port,
COMMAND_LOWER: value 22h, address of lower AIO command port,
DATA_UPPER: value 21h, address of the upper AIO data port,
DATA_LOWER: value 2Øh, address of the lower AIO data port,
INPUT_MODE: value 47h, AIO command to receive input, and
INTERRUPT_DISABLE: value Ø7h, AIO command to disable interrupts.

AIO_INIT uses no module level variables.

c. Routine

AIO_INIT uses the variable NULL (type Byte) as a dummy return variable for the call to IOIN. There are no routine level constants.

8. Other Routines Called

AIO_INIT calls four external assembly language routines, DISABLEINT, ENABLEINT, IOOUT, and IOIN, to accomplish its purpose. These four routines are declaired externals. Descriptions of these routines follow.

a. DISABLEINT

AIO_INIT uses DISABLEINT to disable the Z-8Ø interupts during AIO board initilization. This is a Zilog recommended practice to prevent inadvertant interupts during the initilization. DISABLEINT has no input or output parameters; it is invoked solely by name.

b. ENABLEINT

ENABLEINT is the last routine called by AIO_INIT. It enables the Z-8Ø interupts disable by the earlier call to DISABLEINT. ENABLEINT has no input or output parameters; it is invoked by name only.

c. IOOUT

AIO_INIT uses IOOUT to write commands to the AIO board. IOOUT is invoked via:

IOOUT(IO_PORT, VALUE)

where both IO_PORT and VALUE are of type Byte. IO_PORT passes the address of input/output port to which the eight bit VALUE will be written. For AIO_INIT both IO_PORT and VALUE are passed constants.

d. IOIN

AIO_INIT uses IOIN to read the data registers of the AIO board and clear them of any value. IOIN is invoked by:

VALUE := IOIN(IO_PORT)

where both VALUE and IO_PORT are of type Byte. IO_PORT is the address of the input/output port from which data is read. The return parameter VALUE carries the eight bit value read in from the port.

9. Output of Routine

a. Parameter Passing Schema

AIO_INIT has no output parameters.

b. System Configuration Changes

AIO_INIT produces several changes in the configuration of the system. First, during the program execution, the system interupts are disabled. Second, the AIO board is put into polled mode and the AIO board is ihnibited from issuing interrupts. Last, the AIO board input registers are cleared.

10. Routine Testing

a. Description of Test

No tests were conducted solely on AIO_INIT. Rather, it was tested in conjunction with the other routines which could not function at all if AIO_INIT didn't work.

b. Results of Test

The other routines worked, therefore AIO_INIT works properly.

11. Reference to Listing

The program listing of AIO_INIT is on page 404.

1. Routine Name: **IN_CHAN_SEL**

2. Part of AIO.PLZ.S Module

3. Written in PLZ; two lines of executable code.

4. Synopsis of Routine

This extremely short routine writes to the AIO board Channel Select register the desired channel number. This forces the AIO board to sample the specified input channel and perform an analog to digital conversion. IN_CHAN_SEL uses the external assembly language routine IOOUT to write the value to the AIO board.

5. Routine Relationships Diagram

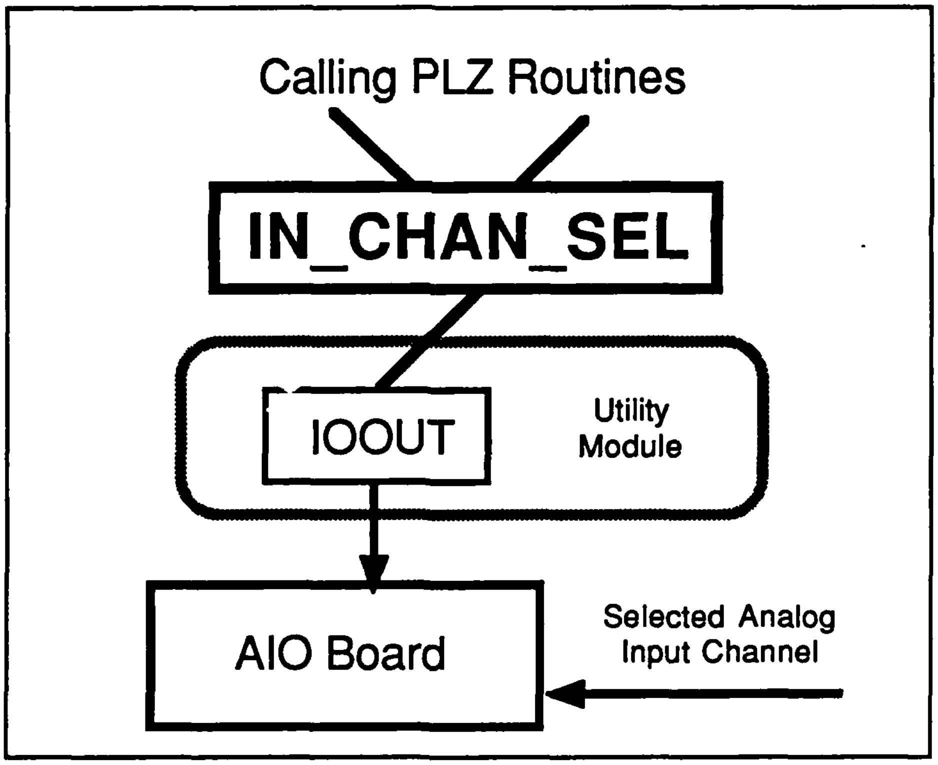


Figure 78. Relationship of IN_CHAN_SEL to Calling PLZ Routine and IOOUT.

6. Invocation

a. Invocation Statement

IN_CHAN_SEL is invoked by:

IN_CHAN_SEL(CHANNEL)

where CHANNEL is of type Byte.

b. Parameter Passing Schema

The input parameter CHANNEL is the number of the analog to digital channel desired.

c. Routines Which Call

IN_CHAN_SEL can be called by any PLZ language routine using the AIO board. The AIO.PLZ.S and UTILITY modules must be linked in with the calling routine. For this thesis effort, IN_CHAN_SEL was used during initial work with the AIO board. For the final thesis effort routines, an pair of interrupt driven assembly language routines, TC_SAMPLE and TO_SAMPLE, perform the channel selection, initiation of analog to digital conversion function.

7. Variables and Constants

a. Global

IN_CHAN_SEL uses no globally defined constants or variables.

b. Module

IN_CHAN_SEL uses the module level constant CHANNEL_SELECT, value 28 hexidecimal, the address of the channel selection register of the AIO board. IN_ CHAN_SEL uses no module level variables.

c. Routine

IN_CHAN_SEL uses no routine level constants and variables.

8. Other Routines Called

IN_CHAN_SEL calls the external assembly language routine IOOUT to write the channel number to the AIO board. IOOUT is invoked by:

IOOUT(IO_PORT, VALUE)

where both IO_PORT and VALUE are of type Byte. IO_PORT is the address of the desired IO port and VALUE is the eight bit to be output.

9. Output of Routine

a. Parameter Passing Schema

IN_CHAN_SEL has no output parameters.

b. System Configuration Changes

IN_CHAN_SEL, by selecting a channel, initiates an analog to digital conversion on the selected channel of the AIO board.

10. Routine Testing

a. Description of Test

IN_CHAN_SEL was tested in conjuction with other routines using the AIO board. If IN_CHAN_SEL didn't work, routine IN_DIGITALP would not find the correct value (from a known, constant input voltage) in the AIO data registers. Several channels were selected by IN_CHAN_SEL and read by IN_DIGITALP.

b. Results of Test

The digital values corresponding to the analog inputs were found by IN_DIGITALP in the AIO data registers.

11. Reference to Listing

The listing of IN_CHAN_SEL can be found on page 404.

1. Routine Name: **IN_DIGITALP**

2. Part of AIO.PLZ.S Module

3. Written in PLZ; four lines of executable code.

4. Synopsis of Routine

IN_DIGITALP reads the data registers of the AIO board to obtain the digital value converted from the analog channel selected by routine IN_CHAN_SEL. IN_DIGITALP loops, polling the AIO status register until an analog to digital conversion is complete. Then IN_DIGITALP reads data from both AIO eight bit data registers and combines them into a single sixteen bit Integer value. Note that AIO analog to digital conversion yields only 12 bits of information. Thus the upper data register holds only four bits of information.

5. Routine Relationship Diagram

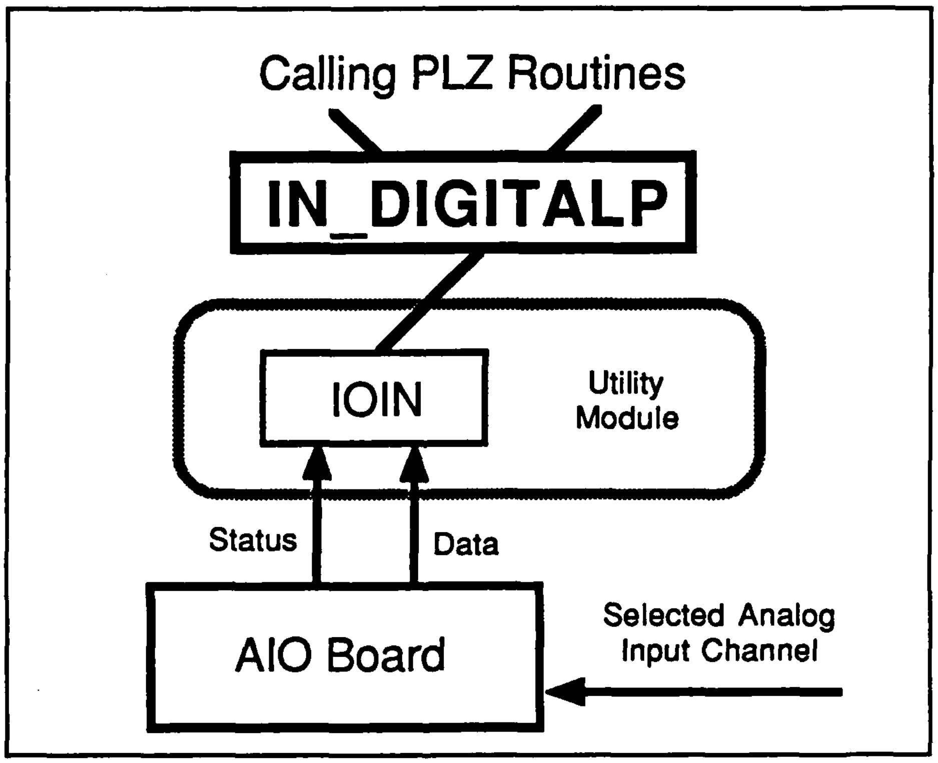


Figure 79. Relationship of IN_DIGITALP to Calling PLZ Routine and IOIN.

6. Invocation

a. Invocation Statement

IN_DIGITALP is invoked from a calling PLZ routine by:

VALUE := IN_DIGITALP

where the return parameter VALUE is of type Integer.

b. Parameter Passing Schema

IN_DIGITALP has no input parameters. The input channel is selected in advance by IN_CHAN_SEL.

c. Routines Which Call

Any PLZ routine needing to obtain analog to digital conversions from the AIO board can use IN_DIGITALP. The AIO.PLZ.S and UTILITY modules must be linked in with the calling routine's module. IN_DIGITALP is not present in the final routines for this thesis effort. IN_DIGITALP was used during initial work to learn how to use the AIO board. . In the final theis effort routines, an interrupt-paced assembly language routine, COLLECTER, is used read data in from the AIO board.

7. Variables and Constants

a. Global

IN_DIGITALP uses no globally defined variables or constants.

b. Module

Four module level constants are used by IN_DIGITALP. Their values and uses are

STATUS:	Value 29 hex, the address of the AIO board status register
MASK:	Value Ø1 hex, a logical masking word to retain only the least significant bit
DATA_UPPER:	Value 21 hex, the address of the upper AIO board data register
DATA_LOWER:	Value 2Ø hex, the address of the lower AIO board data register

IN_DIGITALP uses no module level variables.

c. Routine

IN_DIGITALP uses no routine level variables or constants. The explicit constant 1ØØ hex (represented by %1ØØ) is employed in the combining of the upper and lower data values from the AIO board.

8. Other Routines Called

The external assembly language routine IOIN is used by IN_DIGITALP to both check the AIO board status register and to read in the converted values. IOIN is invoked with:

```
VALUE := IOIN( IO_PORT )
```

where both VALUE and IO_PORT are of type Byte. The input parameter IO_PORT is the address (ØØ hex to FF hex) of the input/output port from which the output parameter VALUE is to be obtained.

9. Output of Routine

a. Parameter Passing Schema

IN_DIGITALP returns to its calling routine a single, type Integer, return para- meter called VALUE. It holds the twelve bit value formed from the upper (four bits) and lower (eight bits) read from the AIO board's two data registers.

b. System Configuration Changes

The configuration of the system is not changed by IN_DIGITALP aside from clearing the AIO board data registers.

10. Routine Testing

a. Description of Test

IN_DIGITALP was tested by having it read from an AIO channel that was fed constant voltages.

b. Results of Test

IN_DIGITALP provided correct digital values to the calling routine.

11. Reference to Listing

The program listing for IN_DIGITALP is on page 405.

1. Routine Name: **IN_DIGITALT**

2. Part of AIO.PLZ.S Module

3. Written in PLZ; three lines of executable code.

4. Synopsis of Routine

IN_DIGITALT is a combination of IN_CHAN_SEL and IN_DIGITALP and con- sists simply of calls to those two routines. Its purpose is to select an AIO channel for input, then wait for the analog to digital conversion to occur, and finally read in the con- verted value. IN_DIGITALT was written for those PLZ programs that:

a. can afford to wait, or

b. do not need to accomplish other tasks during the analog to digital conversion period.

5. Routine Relationships Diagram

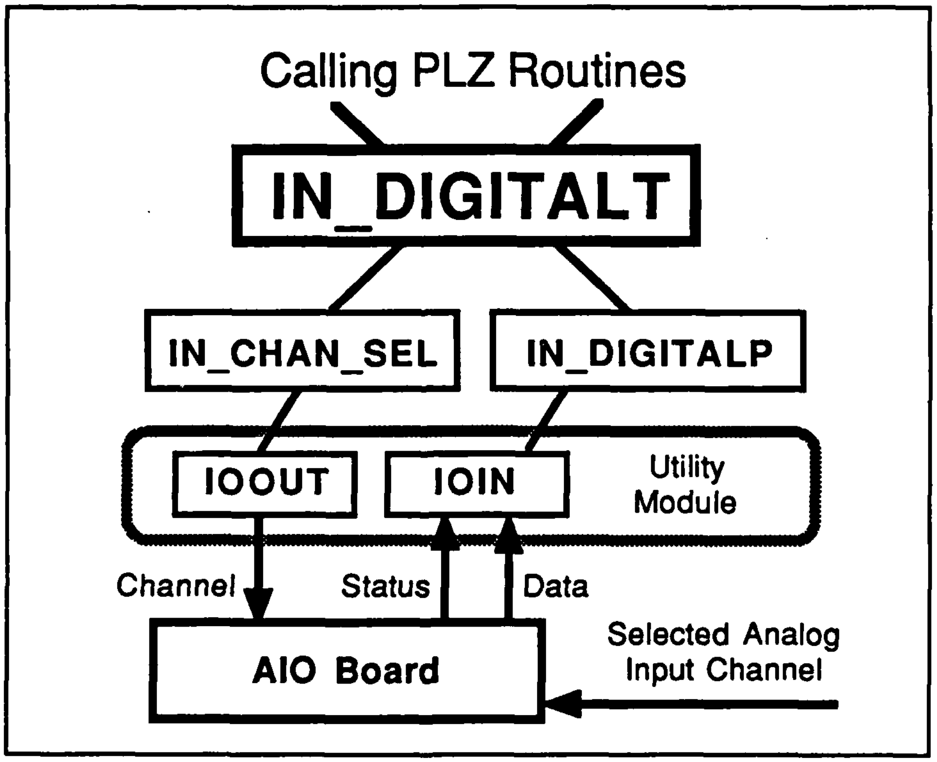


Figure 80. Relationship of IN_DIGITALT to Calling PLZ Routine, IN_CHAN_SEL and IN_DIGITALP.

6. Invocation

a. Invocation Statement

IN_DIGITALT is invoked by:

VALUE := IN_DIGITALT(CHANNEL)

where both VALUE and CHANNEL are of type Byte.

b. Parameter Passing Schema

The single input parameter for IN_DIGITALT, CHANNEL, is the same as for routine IN_DIGITALP, the number of the AIO input channel on which the analog to digital conversion will be made. CHANNEL has a defined range of Ø to F hexidecimal.

c. Routines Which Call

Any PLZ routine needing to get analog-to-digital values from the AIO board can use IN_DIGITALT. To call IN_DIGITALT, both the AIO.PLZ.S and UTILITY modules must be linked in with the calling routine. As with the other routines of the AIO.PLZ.S Module, IN_DIGITALT was used during initial work with the AIO board. IN_DIGITALT does not appear in any of the final programs of this thesis effort.

7. Variables and Constants

IN_DIGITALT uses no variables or constants.

8. Other Routines Called

IN_DIGITALT calls IN_CHAN_SEL to select the analog input channel on the AIO board and IN_DIGITALP to read in the converted digital value from theAIO board.

a. IN_CHAN_SEL initiates an analog to digital conversion on a specific analog input channel. It is invoked via:

IN_CHAN_SEL(CHANNEL)

where the input parameter CHANNEL, type Byte, specifies the desired analog channel. CHANNEL is the input parameter for IN_DIGITALT.

b. IN_DIGITALP reads the converted digital values from the AIO data registers and combines them to form a single integer type value. IN_DIGITALP is invoked by:

VALUE := IN_DIGITALP

where the return parameter VALUE, type Integer, holds the converted, single value. VALUE is then the output parameter for IN_DIGITALT.

9. Output of Routine

a. Parameter Passing Schema

IN_DIGITALT has a single output parameter, VALUE. This sixteen bit parameter passes the twelve bits of information read from the AIO board data registers back to the calling PLZ routine.

b. System Configuration Changes

IN_DIGITALT initiates an analog to digital conversion on a specified AIO input channel. Later, IN_DIGITALT clears the data registers of the AIO board when it reads the converted analog values.

10. Routine Testing

IN_DIGITALT was not tested as it is simply the combination of IN_CHAN_SEL and IN_DIGITALP. Both of these routines were tested and found to function correctly. Testing was considered unnecessary.

11. Reference to Listing

The program listing of IN_DIGITALT is on page 405.

1. Routine Name: **OUT_ANALOG**

2. Part of AIO.PLZ.S Module

3. Written in PLZ; nine lines of executable code.

4. Synopsis of Routine

OUT_ANALOG takes the integer value passed to it, splits the value into two bytes, and outputs the digital values to the AIO board for conversion to an analog signal. OUT_ANALOG can output on either of the two digital to analog channels of the AIO board. The writing of the bytes is accomplished with the external routine IOOUT.

5. Routine Relationships Diagram

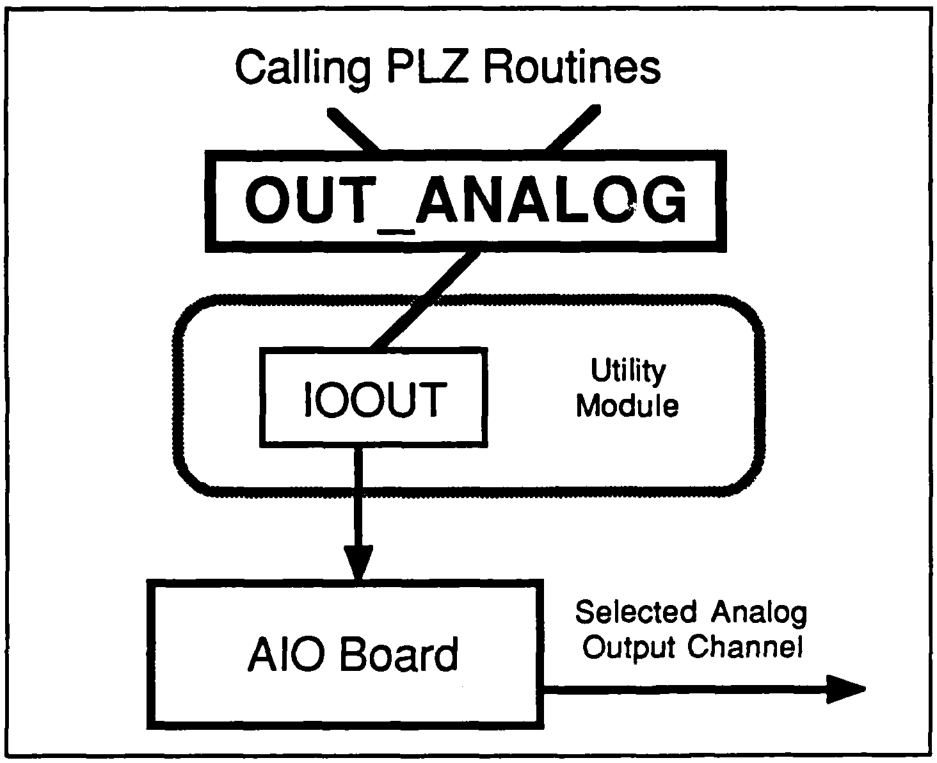


Figure 81. Relationship of OUT_ANALOG to Calling PLZ Routine and IOOUT.

6. Invocation

a. Invocation Statement

OUT_ANALOG is called from a PLZ routine with:

OUT_ANALOG(CHANNEL, VALUE)

where CHANNEL is type Byte and VALUE is type Integer.

b. Parameter Passing Schema

The two input parameters CHANNEL and VALUE pass to OUT_ ANALOG the digital-to-analog channel desired for output and the twelve bits of digital information to be converted to an analog signal by the AIO board. OUT_ ANALOG assumes that VALUE has only twelve significant bits; as an Integer it is a sixteen bit value.

c. Routines Which Call

OUT_ANALOG can be used by any PLZ routine which needs to output analog values. The AIO.PLZ.S and UTILITY modules need to linked in with the calling routine. OUT_ANALOG is not used by any routines of this thesis effort. Like the other PLZ language routines of the AIO.PLZ.S Module it was used for initial investigations of the AIO board. An assembly language version of OUT_ ANALOG, routine OUTDA, was written but is not a part of the final thesis effort routines.

7. Variables and Constants

a. Global

OUT_ANALOG uses no globally defined variables or constatnts.

b. Module

OUT_ANALOG uses four module level constants. Their definitions and values are on the next page.

DA_CHANNEL_1_UPPER: Value 2D hex, IO port address of AIO digital to analog channel one, upper four bit register.

DA_CHANNEL_1_LOWER: Value 2C hex, IO port address of AIO digital to analog channel one, lower eight bit register.

DA_CHANNEL_2_UPPER: Value 2F hex, IO port address of AIO digital to analog channel two, upper four bit register.

DA_CHANNEL_2_LOWER: Value 2E hex, IO port address of AIO digital to analog channel two, lower eight bit register.

These constants are used by OUT_ANALOG when calling IOOUT. OUT_ ANALOG uses no module level variables.

c. Routine

A single routine level constant, OUTVALUE, of type Byte is used by OUT_ ANALOG. It is set to the lower eight bits of the input integer VALUE and is then used to output to the lower data register of the AIO output channel. OUT-VALUE is next set to the upper four bits of the twelve bit input value. OUTVALUE is then used written to the upper data register of the AIO output channel. OUT-ANALOG uses no routine level constants.

8. Other Routines Called

OUT_ANALOG uses two PLZ type conversion functions and one external routine, IOOUT. The type conversions, integer to byte and byte to integer, are used in the splitting of the input parameter VALUE in to the upper four bit and lower eight bit byte values passed to the AIO board via IOOUT. IOOUT is an external assembly language routine of the Utility Module. It permits PLZ language routines direct access to input output ports. IOOUT is invoked via:

IOOUT(IO_PORT, VALUE)

where both IO_PORT and VALUE are of type Byte. IO_PORT is the number or address the input/output port that VALUE is to be written to. IOOUT has no return parameters.

9. Output of Routine

a. Parameter Passing Schema

OUT_ANALOG has no output parameters.

b. System Configuration Changes

OUT_ANALOG sets one of the digital to analog channels to a value. That value will continue to be output by the analog channel until either another value is written to it or the AIO board is turned off.

10. Routine Testing

a. Description of Test

OUT_ANALOG was tested through a looping routine which read in an analog to digital conversion value, via IN_DIGITALT, and then output that value back through OUT_ANALOG. A low frequency sine wave input was applied to the analog input. Both the sine input and the output of the digital to analog channel were monitored by an oscilliscope.

b. Results of Test

The output channel tracked the input channel with the time delay produced by the processing delay.

11. Reference to Listing

The listing of OUT_ANALOG is on page 406.

Program Listings of AIO.PLZ.S Module

The following pages are a listing of the AIO.PLZ.S Module routines This is not a compiled or assembled listing; it is PLZ source code.

```
AIO.PLZ.S MODULE

        !       AIO.PLZ.S is a collection of PLZ language routines for the
        employement of the AIO analog input and output board.  These routines
        handle the initilization of the board (ANALOGINIT), analog to digital
        conversion and input ( INDIGITALT, INCHANSEL, INDIGITP ), and digital
        to analog output ( OUTANALOG ).  These five routines and the calling
        PLZ routines pass the channel number and I/O values between each other.
        These routines require the assembly language routines IOIN and IOOUT
        to be linked in.                                                       !

CONST
        COMMAND_UPPER           := %23
        COMMAND_LOWER           := %22
        DATA_UPPER              := %21
        DATA_LOWER              := %20
        CHANNEL_SELECT          := %28
        STATUS                  := %29
        DA_CHANNEL_1_UPPER      := %2D
        DA_CHANNEL_1_LOWER      := %2C
        DA_CHANNEL_2_UPPER      := %2F
        DA_CHANNEL_2_LOWER      := %2E
        MASK                    := %01
        INPUT_MODE              := %47
        INTERRUPT_DISABLE       := %07

EXTERNAL

        IOOUT PROCEDURE (PORT OUTVALUE  BYTE)

        IOIN  PROCEDURE (PORT  BYTE)
              RETURNS   (INVALUE  BYTE)

        ENABLEINT PROCEDURE

        DISABLEINT PROCEDURE


INTERNAL

        NULL  BYTE
```

```
AIO_INIT PROCEDURE

        !       This procedure sets the AIO board into polled mode for
        the analog to digital converters.  It also clears the input
        data registers by reading from them.                        !

ENTRY

  DISABLEINT

  IOOUT(COMMAND_UPPER,INPUT_MODE)
  IOOUT(COMMAND_LOWER,INPUT_MODE)
  IOOUT(COMMAND_UPPER,INTERRUPT_DISABLE)
  IOOUT(COMMAND_LOWER,INTERRUPT_DISABLE)

  NULL := IOIN(DATA_LOWER)
  NULL := IOIN(DATA_UPPER)

  ENABLEINT

  END AIO_INIT


IN_CHAN_SEL  PROCEDURE(CHANNEL BYTE)

!       IN_CHAN_SEL outputs to the channel select register the
number of the register for which the next analog to digital
conversion is to occur.  This action initiates the d to a
conversion.    Channels 0 through 15 are defined.            !


ENTRY
  IOOUT(CHANNEL_SELECT,CHANNEL)
  END IN_CHAN_SEL
```

```
INDIGITALP PROCEDURE
           RETURNS  (VALUE INTEGER)

!       INDIGITALP is a routine to read the converted analog signal
into the program.  This routine assumes that IN_CHAN_SEL has already
been called to initiate the conversion and itentify the desired
channel.                                                          !

ENTRY
  DO
    IF IOIN(STATUS) AND MASK THEN EXIT FI
    OD
  VALUE := INTEGER IOIN(DATA_UPPER)
  VALUE := VALUE * %100 + INTEGER IOIN(DATA_LOWER)
  END INDIGITALP


INDIGITALT PROCEDURE( CHANNEL BYTE )
           RETURNS  ( VALUE INTEGER )

!       This procedure combines the operations of IN_CHAN_SEL
and INDIGITALP.  Calling this single routine will require somewhat
longer conversion times.  For those applications requireing faster
response the calling PLZ routine should call IN_CHAN_SEL prior
to its need of the data.  Then the routine should call INDIGITALP
to obtain the data.  As with IN_CHAN_SEL, channels 0 through 15
are defined.                                                   !

ENTRY
  IN_CHAN_SEL( CHANNEL )
  VALUE := INDIGITALP( CHANNEL )
  END INDIGITALT
```

```
        OUT_ANALOG PROCEDURE( CHANNEL BYTE, VALUE INTEGER )

        !       This routine outputs the analog to conversion value.
        The analog to digital converters are 12 bit devices so the
        calling routine cannot expect greater that 12 bit accuracy.
        Only channels 1 and 2 exit.  Calling this routine with any
        other channel number will result in output on channel 1.  !


        ENTRY
          OUTVALUE := BYTE VALUE
          IF CHANNEL=1
            THEN IOOUT(DA_CHANNEL_1_LOWER,OUTVALUE)
            ELSE IOOUT(DA_CHANNEL_2_LOWER,OUTVALUE)

            FI
          OUTVALUE := (VALUE - INTEGER OUTVALUE)/%100
          IF CHANNEL=1
            THEN IOOUT(DA_CHANNEL_1_UPPER,OUTVALUE)
            ELSE IOOUT(DA_CHANNEL_2_UPPER,OUTVALUE)

          END OUT_ANALOG

END AIO.PLZ.S
```

Appendix H: Scale_Factor Module

Scale_Factor Module is a compiled set of PLZ language routines which implement the Set Up Scale Factor File process shown in Figure 3 in the introduction. With the routines within Scale Factor, a user can create or modify a disk file of scale factors. Due to the difficulties encountered in debugging the Collect and Store Data process routines, Scale_Factor Module was not integrated in with the other software of this thesis effort.

The routines of Scale_Factor Module are listed here to show how the IO improvements of the Enhancements Module can be used. The module is organized into an executive / subordinate routine structure as shown by Figure 82 below. The module executes the subordinate routines in sequence. The most complex of the subordinate routines, CHANGE_SCALE, makes extensive use of the Enhanacements Module routines. The program flow within CHANGE_SCALE is shown in Figure 83. Both figures are present to aid reader understanding of the execution of Scale_Factor Module.

The listings of Scale_Factor Module routines are on the following pages.

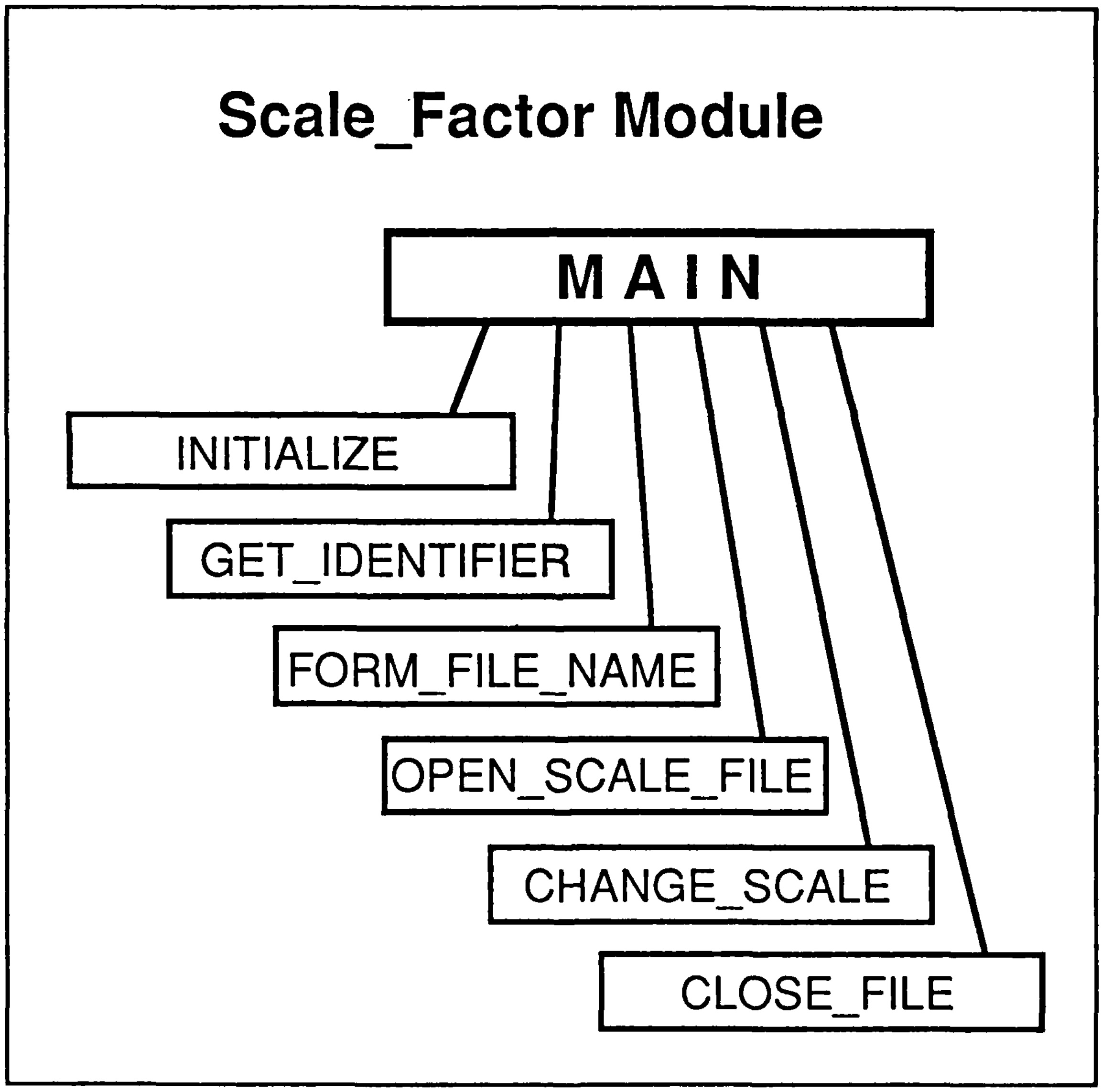


Figure 82. Hierarchical Organization of Scale_Factor Module

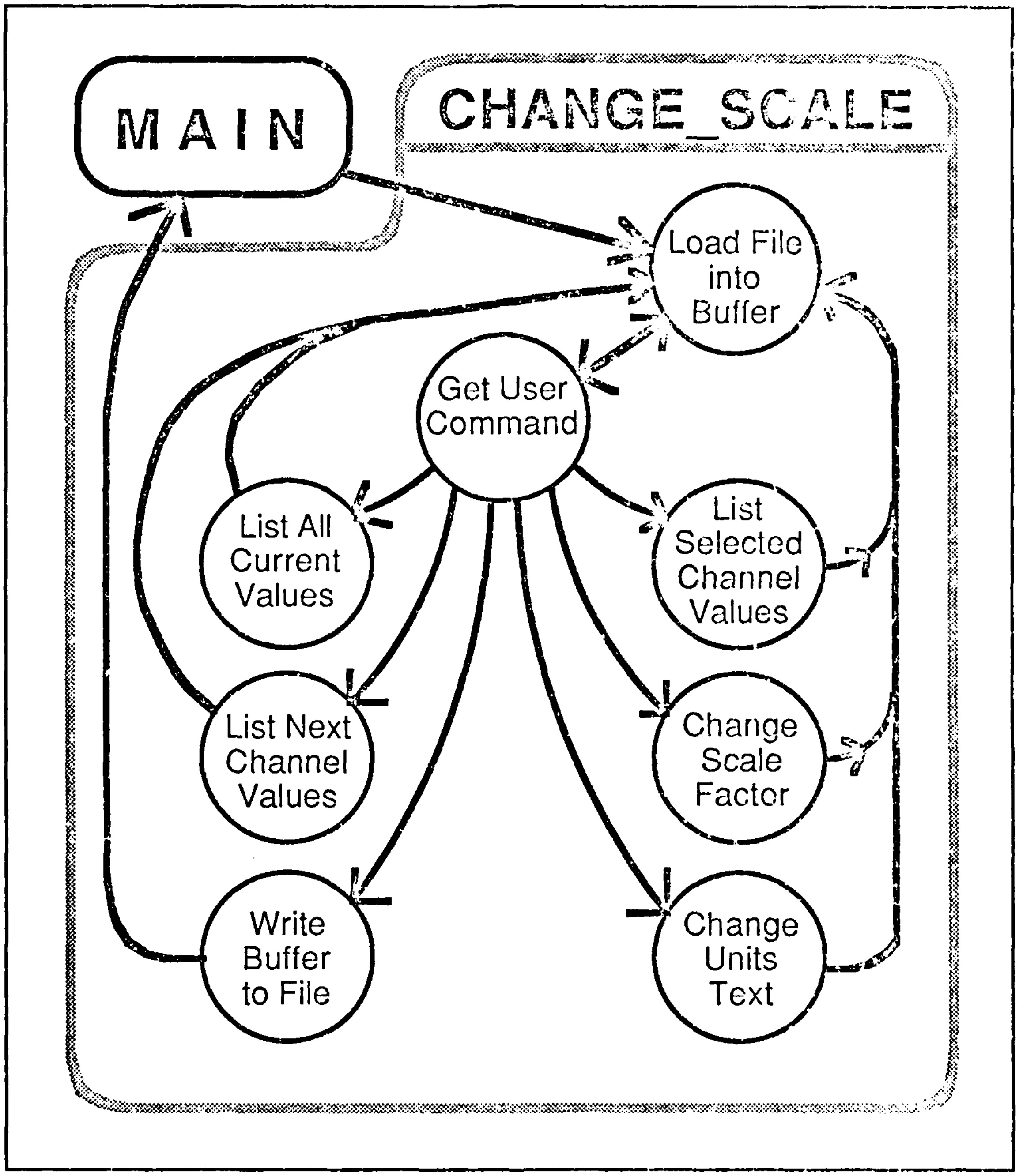


Figure 83. Program Execution Flow Within CHANGE_SCALE

PLZSYS 3.0 801121-1

```
SCALE_FACTOR    MODULE      ! 21 November 1980 !

CONSTANT

  TRUE  := 1
  FALSE := 0
  BLANK := ' '
  CR_RETURN  := %0D
  BACKSPACE  := %08
  OPEN_INPUT := 0
  OPEN_NEWFILE := 2
  FILE_NOT_FOUND := %C7
  SCALE_FILE := 11
  CONSOLE_IN  := 1
  CONSOLE_OUT := 2



TYPE

  PBYTE ^BYTE



EXTERNAL

  OPEN   PROCEDURE ( LOGICAL_UNIT BYTE, FILE_NAME PBYTE, FLAG BYTE )
         RETURNS   ( RETURN_CODE BYTE )
  CLOSE  PROCEDURE ( LOGICAL_UNIT BYTE )
         RETURNS   ( RETURN_CODE BYTE )
  GETSEQ PROCEDURE ( LOGICAL_UNIT BYTE, BUFFER_PTR PBYTE, NUMBER_BYTES WORD )
         RETURNS   ( RETURN_BYTES WORD, RETURN_CODE BYTE )
  PUTSEQ PROCEDURE ( LOGICAL_UNIT BYTE, BUFFER_PTR PBYTE, NUMBER_BYTES WORD )
         RETURNS   ( RETURN_BYTES WORD, RETURN_CODE BYTE )
  SEEK   PROCEDURE ( LOGICAL_UNIT BYTE, POSHIGH WORD, POSLOW WORD, FLAG BYTE )
         RETURNS   ( RETURN_CODE BYTE )
  WRITE_RCODE PROCEDURE( RETURN_CODE BYTE )



GLOBAL  ! INTERNAL !

  FILE_NAME    ARRAY [ 32 BYTE ]
  IDENTIFIER   ARRAY [  9 BYTE ]
  INPUT_BUF    ARRAY [ 48 BYTE ]
  DIGIT        ARRAY [ 11 BYTE ]
  PDIGIT      PBYTE



  INITIALIZE PROCEDURE

    ENTRY
```

```
        DIGIT[  0 ] := '0'
        DIGIT[  1 ] := '1'
        DIGIT[  2 ] := '2'
        DIGIT[  3 ] := '3'
        DIGIT[  4 ] := '4'
        DIGIT[  5 ] := '5'
        DIGIT[  6 ] := '6'
        DIGIT[  7 ] := '7'
        DIGIT[  8 ] := '8'
        DIGIT[  9 ] := '9'
        DIGIT[ 10 ] := BLANK
        PDIGIT := #DIGIT[0]
      END INITIALIZE



    WRITELN PROCEDURE( LOGICAL_UNIT BYTE, TEXT_PTR PBYTE )

      LOCAL
        LENGTH WORD
        RETURN_CODE  BYTE
        PINDEX PBYTE
      ENTRY
        LENGTH := 1
        PINDEX := TEXT_PTR
        DO
          IF PINDEX^ = CR_RETURN THEN EXIT FI
          LENGTH := LENGTH + 1
          PINDEX := INC PINDEX
        OD
        LENGTH, RETURN_CODE := PUTSEQ( LOGICAL_UNIT, TEXT_PTR, LENGTH )
      END WRITELN



    READLN PROCEDURE( LOGICAL_UNIT BYTE, STRING_BEGINNING PBYTE )

      LOCAL
        LENGTH      WORD
        RETURN_CODE BYTE
        STRING_PTR  PBYTE
      ENTRY
        STRING_PTR := STRING_BEGINNING
        DO
          LENGTH, RETURN_CODE := GETSEQ( LOGICAL_UNIT, STRING_PTR, 1 )
          IF STRING_PTR^ = CR_RETURN THEN EXIT FI
          STRING_PTR := INC STRING_PTR
        OD
      END READLN



    WRITE PROCEDURE( LOGICAL_UNIT BYTE, TEXT_PTR PBYTE )

```

```
          LOCAL
            LENGTH WORD
            RETURN_CODE  BYTE
            PINDEX PBYTE
          ENTRY
1           LENGTH := 0
2           PINDEX := TEXT_PTR
            DO
3             IF PINDEX^ = CR_RETURN THEN EXIT  FI
4             LENGTH := LENGTH + 1
5             PINDEX := INC PINDEX
            OD
6           LENGTH, RETURN_CODE := PUTSEQ( LOGICAL_UNIT, TEXT_PTR, LENGTH )
7         END WRITE



        READ_CH PROCEDURE( LOGICAL_UNIT  BYTE )
                RETURNS  ( CH BYTE )
          LOCAL
            RBYTES      WORD
            RETURN_CODE BYTE
          ENTRY
            DO
1             RBYTES, RETURN_CODE := GETSEQ( LOGICAL_UNIT, #CH, 1 )
            OD
2         END READ_CH



        WRITE_CH PROCEDURE( LOGICAL_UNIT BYTE, CHARACTER PBYTE )

-         LOCAL
            RBYTES      WORD
            RETURN_CODE BYTE
          ENTRY
1           RBYTES, RETURN_CODE := PUTSEQ( LOGICAL_UNIT, CHARACTER, 1 )
2         END WRITE_CH



        ACCEPTABLE PROCEDURE ( CHARACTER  BYTE )
                   RETURNS   ( ACCEPTABLE BYTE )
          ENTRY
1           ACCEPTABLE := TRUE
2           IF CHARACTER > %7A ORIF  CHARACTER < %30  ORIF
             ( CHARACTER > %5A ANDIF CHARACTER < %61 ) ORIF
             ( CHARACTER > %39 ANDIF CHARACTER < %41 ) THEN
3             ACCEPTABLE := FALSE FI
4           IF CHARACTER = %0D THEN ACCEPTABLE := TRUE FI
6         END ACCEPTABLE
```

```
       GET_IDENTIFIER  PROCEDURE( TEST_ID PBYTE )

         LOCAL
           INDEX               INTEGER
           CHARACTER READY     BYTE
         ENTRY
 1         READY := FALSE
           DO
 2           INDEX := 0
             DO
 3             IF TEST_ID^ <> BLANK  THEN EXIT FI
 4             TEST_ID := INC TEST_ID
             OD
             DO
 5             IF ACCEPTABLE( TEST_ID^ ) = TRUE  THEN
 6               IDENTIFIER[ INDEX ] := TEST_ID^
 7               IF TEST_ID^ = CR_RETURN  THEN
 8                 IF INDEX > 0 THEN READY := TRUE FI
                   EXIT
                   FI
10               INDEX := INDEX + 1
11               IF INDEX > 7 THEN
12                 IDENTIFIER[ INDEX ] := CR_RETURN
13                 READY := TRUE
14                 EXIT
                   FI
                 FI
15             TEST_ID := INC TEST_ID
             OD
16           IF READY = TRUE THEN EXIT FI
17           INDEX := 0
18           WRITELN( CONSOLE_OUT, $'Enter test ID, 8 characters max %R')
19           TEST_ID := $INPUT_BUF[0]
20           READLN( CONSOLE_IN, TEST_ID )
           OD
21         WRITELN( CONSOLE_OUT, $FILE_NAME[0] )       ! DEBUG **************** !
22     END GET_IDENTIFIER


       FORM_FILE_NAME PROCEDURE

         LOCAL
           INDEX INTEGER
         ENTRY
 1         FILE_NAME[ 0 ] := 'S'
 2         FILE_NAME[ 1 ] := 'C'
 3         FILE_NAME[ 2 ] := 'A'
 4         FILE_NAME[ 3 ] := 'L'
 5         FILE_NAME[ 4 ] := 'E'
 6         FILE_NAME[ 5 ] := '.'
 7         INDEX := 0
           DO
 8           IF IDENTIFIER[ INDEX ] = CR_RETURN THEN EXIT FI
```

```
        FILE_NAME[ INDEX + 6 ] := IDENTIFIER[ INDEX ]
        INDEX += 1
      OD
      FILE_NAME[ INDEX + 6 ] := '.'
      FILE_NAME[ INDEX + 7 ] := '\'
      FILE_NAME[ INDEX + 8 ] := CR_RETURN
      WRITELN( CONSOLE_OUT, #FILE_NAME[0] )        ! DEBUG ******************!
    END FORM_FILE_NAME



  CREATE_SCALE_FILE  PROCEDURE( FILE_UNIT BYTE, FILE_NAME_PTR PBYTE )

    LOCAL
      INDEX     INTEGER
      RETURN_CODE   BYTE
    ENTRY
      INDEX := 0
      WRITELN( CONSOLE_OUT, #' create_scale_file procedure %R' )
      RETURN_CODE := OPEN( FILE_UNIT, FILE_NAME_PTR, OPEN_NEWFILE )
      WRITE( CONSOLE_OUT, #'open return_code = %R' )
      WRITE_RCODE( RETURN_CODE )
      DO
        WRITE( FILE_UNIT, #'Channel # %R' )
        IF INDEX < 10
          THEN
            WRITE( FILE_UNIT, #' %R' )
            WRITE_CH( FILE_UNIT, #DIGIT[ INDEX ] )
          ELSE
            WRITE( FILE_UNIT, #'1%R' )
            WRITE_CH( FILE_UNIT, #DIGIT[ INDEX - 10 ] )
          FI
        WRITE( FILE_UNIT, #': %R' )
        WRITE( FILE_UNIT, #'  S= __1.000 E 00 %R' )
        WRITELN( FILE_UNIT, #'U=_____________%R' )
        IF INDEX = 15 THEN EXIT FI
        INDEX := INDEX + 1
      OD
    END CREATE_SCALE_FILE



  OPEN_SCALE_FILE PROCEDURE

    LOCAL
      RETURN_CODE   BYTE
    ENTRY
      WRITELN( CONSOLE_OUT, #' open_scale_file procedure %R' )
      WRITE( CONSOLE_OUT, #' file name : %R' )
      WRITE( CONSOLE_OUT, #FILE_NAME[0] )
      RETURN_CODE := OPEN( SCALE_FILE, #FILE_NAME[0], OPEN_INPUT )
      WRITE( CONSOLE_OUT, #'open_scale_file open return_code = %R' )
      WRITE_RCODE( RETURN_CODE )
      IF RETURN_CODE = FILE_NOT_FOUND  THEN CREATE_SCALE_FILE( SCALE_FILE, #FILE_NAME[0] ) FI
```

```
        RETURN_CODE := SEEK( SCALE_FILE, 0, 0, 0 )
        WRITE( CONSOLE_OUT, #'open_scale_file seek return_code = %R' )
        WRITE_RCODE( RETURN_CODE )
      END OPEN_SCALE_FILE



    NEW_SCALER PROCEDURE( LOCATION PBYTE, INPUT_STRING PBYTE )

      LOCAL
        CHARSET ARRAY [ 4 BYTE ]
        LEADING_BLANKS  COUNT  INDEX  INTEGER
        CHARACTER1  CHARACTER2  BYTE
      ENTRY
        INDEX := 0
        DO
          IF INDEX = 18 THEN EXIT FI
          LOCATION := INC LOCATION
          INDEX    := INDEX + 1
        OD
        DO
          IF INPUT_STRING^ = '+' ORIF INPUT_STRING^ = '-' ORIF INPUT_STRING^ = BLANK THEN EXIT FI
          INPUT_STRING := INC INPUT_STRING
        OD
        LOCATION^ := INPUT_STRING^
        LOCATION := INC LOCATION
        INDEX    := INDEX + 1
        CHARSET[ 0 ] := INPUT_STRING^
        COUNT := 0
        DO
          IF CHARSET[ COUNT ] = '.' THEN EXIT FI
          IF CHARSET[ COUNT ] >= %30 ANDIF CHARSET[ COUNT ] <= %39  THEN
            IF COUNT = 3  THEN CHARSET[ 3 ] := '.'  FI
          COUNT := COUNT + 1
          FI
          INPUT_STRING := INC INPUT_STRING
          CHARSET[ COUNT ] := INPUT_STRING^
        OD
        LEADING_BLANKS := 3 - COUNT
        DO
          IF LEADING_BLANKS = 0 THEN EXIT FI
          LOCATION^ := BLANK
          LOCATION := INC LOCATION
          INDEX    := INDEX + 1
          LEADING_BLANKS := LEADING_BLANKS - 1
        OD
        COUNT := 0
        DO
          LOCATION^ := CHARSET[ COUNT ]
          LOCATION := INC LOCATION
          INDEX    := INDEX + 1
          IF CHARSET[ COUNT ] = '.'  THEN EXIT FI
          COUNT := COUNT + 1
        OD
```

```
        INPUT_STRING := INC INPUT_STRING
        COUNT := 0
        DO
          IF INPUT_STRING^ < %30  ORIF  INPUT_STRING^ > %39  ORIF  COUNT = 3  THEN EXIT FI
          LOCATION^ := INPUT_STRING^
          LOCATION := INC LOCATION
          INDEX    := INDEX + 1
          COUNT := COUNT + 1
          INPUT_STRING := INC INPUT_STRING
        OD
        DO
          IF COUNT = 3 THEN EXIT FI
          LOCATION^ := '0'
          LOCATION := INC LOCATION
          INDEX    := INDEX + 1
          COUNT := COUNT + 1
        OD

        DO
          IF INDEX = 27 THEN EXIT FI
          LOCATION := INC LOCATION
          INDEX    := INDEX + 1
        OD
        DO
          IF INPUT_STRING^ = 'E' THEN EXIT FI
          INPUT_STRING := INC INPUT_STRING
        OD
        LOCATION^ := 'E'
        LOCATION := INC LOCATION
        INDEX    := INDEX + 1
        INPUT_STRING := INC INPUT_STRING
        DO
        IF INPUT_STRING^ = '+' ORIF INPUT_STRING^ = '-' ORIF INPUT_STRING^ = BLANK THEN EXIT FI
          INPUT_STRING := INC INPUT_STRING
        OD
        LOCATION^ := INPUT_STRING^
        LOCATION := INC LOCATION
        INDEX    := INDEX + 1
        DO
          IF INPUT_STRING^ >= %30 ANDIF INPUT_STRING^ <= %39 THEN EXIT FI
          INPUT_STRING := INC INPUT_STRING
        OD
        CHARACTER1 := INPUT_STRING^
        INPUT_STRING := INC INPUT_STRING
        CHARACTER2 := INPUT_STRING^
        IF CHARACTER2 >= %30 ANDIF CHARACTER2 <= %39
          THEN
            LOCATION^ := CHARACTER1
            LOCATION := INC LOCATION
            INDEX    := INDEX + 1
            LOCATION^ := CHARACTER2
          ELSE
            LOCATION^ := '0'
```

```
70              LOCATION := INC LOCATION
71              INDEX    := INDEX + 1
72              LOCATION^ := CHARACTER1
              FI
73      END NEW_SCALER


        CHANGE_SCALE PROCEDURE

          LOCAL
            WORK_SPACE ARRAY [ 1024 BYTE ]
            BEGINNING  INPUT_STRING  PBYTE
            CHANNEL  LINE  INDEX     INTEGER
            CHARACTER1  CHARACTER2  RETURN_CODE  BYTE
            RBYTES        WORD
          ENTRY
 1          BEGINNING := #WORK_SPACE[0]
 2          RBYTES, RETURN_CODE := GETSEQ( SCALE_FILE, BEGINNING, 1024 )
 3          WRITE( CONSOLE_OUT, #'change_scale getseq return code = %R' )
 4          WRITE_RCODE( RETURN_CODE )
 5          CHANNEL := 0
            DO
 6            WRITELN( CONSOLE_OUT, #'select command: A, N, O, 0 to 15, S=, or U=.%R' )
 7            WRITELN( CONSOLE_OUT, #'                   All list, Next list, Quit, 0-15 channel number, %R' )
 8            WRITELN( CONSOLE_OUT, #'                   S=(+, ,-)mmn.mmn E=(+, ,-)ee, or U=__________________%R' )
 9            WRITELN( CONSOLE_OUT, #'command :%R' )
10            READLN( CONSOLE_IN, INPUT_STRING )
11            IF INPUT_STRING^ = 'A'  ORIF  INPUT_STRING^ = 'E'  ORIF  INPUT_STRING^ = 'N'  ORIF
                 INPUT_STRING^ = 'O'  ORIF  INPUT_STRING^ = 'S'  ORIF
               ( INPUT_STRING^ >= %30 ANDIF INPUT_STRING^ <= %39 )
12             THEN IF INPUT_STRING^

                CASE 'A'  THEN
13                WRITELN( CONSOLE_OUT, #'list all selected %R' )   ! DEBUG *********!
14                CHANNEL := 0
                  DO
15                  WRITELN( CONSOLE_OUT, #WORK_SPACE[ CHANNEL * 48 ] )
16                  CHANNEL := CHANNEL + 1
17                  IF CHANNEL > 15 THEN EXIT FI
                  OD
                ! end case 'A' !

                CASE 'N'  THEN
18                WRITELN( CONSOLE_OUT, #'list next selected %R' )   ! DEBUG ****** !
19                CHANNEL := CHANNEL + 1
20                IF CHANNEL > 15 THEN CHANNEL := 0 FI
22                WRITELN( CONSOLE_OUT, #WORK_SPACE[ CHANNEL * 48 ] )
                ! end case 'N' !

                CASE 'O'  THEN
23                WRITELN( CONSOLE_OUT, #'quit selected %R' )  ! DEBUG *********** !
24                EXIT
                ! end case 'Q' !
```

```
      CASE 'S'  THEN
        WRITELN( CONSOLE_OUT, #'change of scale factor selected %R' )  ! DEBUG *********** !
        LINE := CHANNEL * 48
        NEW_SCALER( #WORK_SPACE[ LINE ], INPUT_STRING )
        WRITELN( CONSOLE_OUT, #WORK_SPACE[ LINE ] )
      ! end case 'S' !

      CASE 'U'  THEN
        WRITELN( CONSOLE_OUT, #'change of units string selected %R' )  ! DEBUG ************ !
        LINE := CHANNEL * 48
        INDEX := 34
        INPUT_STRING := INC INPUT_STRING
        DO
          IF INDEX >= 46 THEN EXIT FI
          INPUT_STRING := INC INPUT_STRING
          IF INPUT_STRING^ >= %20 THEN
            WORK_SPACE[ LINE + INDEX ] := INPUT_STRING^
            INDEX := INDEX + 1
          FI
        OD
        WRITELN( CONSOLE_OUT, #WORK_SPACE[ CHANNEL * 48 ] )
      ! end case 'U' !

      CASE '0','1','2','3','4','5','6','7','8','9'  THEN
        WRITELN( CONSOLE_OUT, #'list of specific channel selected %R' )   ! DEBUG *************** !
        CHANNEL := 0
        CHARACTER1 := INPUT_STRING^
        IF INPUT_STRING^ = '0' ORIF INPUT_STRING^ = '1' THEN
          INPUT_STRING := INC INPUT_STRING
          CHARACTER2 := INPUT_STRING^
          IF CHARACTER2 >= %30 ANDIF CHARACTER2 <= %39 THEN
            IF CHARACTER1 = 1 ANDIF CHARACTER2 <= %35  THEN CHANNEL := 10 FI
            CHARACTER1 := CHARACTER2
            FI
          FI
        CHANNEL := CHANNEL + INTEGER ( CHARACTER1 - %30 )
        WRITELN( CONSOLE_OUT, #WORK_SPACE[ CHANNEL * 48 ] )
      ! end case '0','1',.....'9' !

      FI ! case statements !

    FI
  DO
    IF INPUT_STRING^ = CR_RETURN THEN EXIT FI
    INPUT_STRING^ := READ_CH( CONSOLE_IN )
  OD

OD
RETURN_CODE := SEEK( SCALE_FILE, 0, 0, 0 )
WRITE( CONSOLE_OUT, #'change_scale seek return code = %R' )
WRITE_RCODE( RETURN_CODE )
RBYTES, RETURN_CODE := PUTSEQ( SCALE_FILE, BEGINNING, 1024 )
WRITE( CONSOLE_OUT, #'change_scale putseq return code = %R' )
WRITE_RCODE( RETURN_CODE )
```

```
59        END CHANGE_SCALE



        CLOSE_FILE PROCEDURE( LOGICAL_UNIT  BYTE )

          LOCAL
            RETURN_CODE  BYTE
          ENTRY
 1          RETURN_CODE := CLOSE( LOGICAL_UNIT )
 2          WRITE( CONSOLE_OUT, #'close_file close return code = %R' )
 3          WRITE_RCODE( RETURN_CODE )
 4        END CLOSE_FILE



      !++++++++++ end of internals ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++



      !  GLOBAL  !



      MAIN PROCEDURE( INTEXT_PTR PBYTE )

        ENTRY
 1        INITIALIZE
 2        GET_IDENTIFIER( INTEXT_PTR )
 3        FORM_FILE_NAME
 4        OPEN_SCALE_FILE
 5        CHANGE_SCALE
 6        CLOSE_FILE( SCALE_FILE )
 7      END MAIN


      END SCALE_FACTOR


END OF COMPILATION:    0 ERROR(S)       0 WARNING(S)
  833 DATA BYTES   1509 Z-CODE BYTES    SYMBOL TABLE 12% FULL
```

Vita

Lloyd Edwin Lutz Jr. was born on 7 November 1951 in Marion, Ohio. He attented high school in Sidney Ohio and graduated in 1970. In March 1975 he graduated from The Ohio State University, receiving a Bachelor of Science in Electrical Engineering degree. Following graduation, he was commissioned into the US Air Force though ROTC. In August 1975 Lloyd E. Lutz Jr. entered active duty at the Air Force Weapons Laboratory, Kirtland AFB, New Mexico, as Program Manager for Satellite Systems Support in the Analysis Division. He entered the School of Engineering, Air Force Institute of Technology in June 1979. Beginning in April 1981 he served as a Staff Scientist in the Electronics Vulnerability Divison of the Defense Nuclear Agency, Washington, DC. In June 1984 Lloyd E. Lutz Jr was assigned to the Electronics Systems Divison, Operations Analysis Directorate, of the Air Force Operational Test and Evaluation Center, Kirtland AFB, New Mexico.

Permanent Address: 2800 West Russell Road
Sidney, Ohio 45365

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) AFIT/GE/ENG/86M-1	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION School of Engineering Air Force Institute of Tech 6b. OFFICE SYMBOL (If applicable) AFIT/ENG	7a. NAME OF MONITORING ORGANIZATION
6c. ADDRESS (City, State and ZIP Code) Air Force Institute of Technology Wright-Patterson AFB, OH 45433	7b. ADDRESS (City, State and ZIP Code)
8a. NAME OF FUNDING/SPONSORING ORGANIZATION 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State and ZIP Code)	10. SOURCE OF FUNDING NOS. PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT NO.
11. TITLE (Include Security Classification) See box 19	

12. PERSONAL AUTHOR(S)
Lloyd E. Lutz Jr., Captain, USAF

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Yr., Mo., Day)	15. PAGE COUNT
MS Thesis	FROM ______ TO ______	1986 February	444

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB. GR.	
09	02		Data Acquisition, Analog to Digital Converters, Digital Computers, Data Storage Systems

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

Title: DESIGN AND PARTIAL IMPLEMENTATION OF A COMPUTER CONTROLLED DATA COLLECTION SYSTEM

Thesis Chairman: Dr. Gary B. Lamont
Professor of Electrical Engineering

Approved for public release: IAW AFR 190-1.
Lynn E. Wolaver 9 May 86
LYNN E. WOLAVER
Dean for Research and Professional Development
Air Force Institute of Technology (ATC)
Wright-Patterson AFB OH 45433

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT. ☐ DTIC USERS ☐	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL Dr. Gary B. Lamont	22b. TELEPHONE NUMBER (Include Area Code) (513)-255-2057 22c. OFFICE SYMBOL AFIT/ENG

DD FORM 1473, 83 APR EDITION OF 1 JAN 73 IS OBSOLETE.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

A computer controlled data collection system was designed and partially implemented in software. The design concept is for a data collection unit to be placed inside the system being tested where it stores the test data in an internal memory. Post-test this internal unit is connected to and polled by an external control and data storage unit which archives the data. Both units are computers. This combination of an internal data collection unit and an external control and storage unit is intended for testing applications where it is either undesireable or not possible to connect the system being tested to external data recording devices during the test event.

The partial implementation of this dual unit data collection system design was performed on a Zilog MCZ Z-80 development system in PLZ, a Pascal-like language, and Z-80 assembly language. Routines to improve the input/output and hardware access of PLZ were written and used. The software to implement the internal data collection unit and portions of the external control and data storage unit were also written. The internal unit routines employ a Zilog Counter Timer Circuit to generate sampling period interrupts. The analog to digital conversion is accomplished via a Zilog Analog Input Output (AIO) board. The data collection system is not fully operational.

END

11-86

DTIC