1-1-1982

A high level input/output system for microprocessor based instruments.

Bruce Allen Muschlitz

A HIGH LEVEL INPUT/OUTPUT SYSTEM FOR MICROPROCESSOR BASED INSTRUMENTS

by

Bruce Allen Muschlitz

A Thesis
Presented to the Graduate Committee
of Lehigh University
in Candidacy for the Degree of
Master of Science

in
Electrical Engineering

Lehigh University
1982

Certification

This Thesis is accepted and approved

in partial fulfillment of the requirements

for the degree of

Master of Science

in

Electrical Engineering

13 May 1982
(date)

Professor in Charge

Chairman of Department

Acknowledgments

The author would like to thank Dr. Authur I. Larky, professor of Electrical and Computer Engineering, for his tireless advice throughout this thesis project. I would also like to thank the Lehigh University ECE department and Carpenter Technology Corporation for their financial support. Finally, I would like thank my devoted fiancee, Darlene, for her patience and understanding during my graduate work.

Table of Contents

List of Figures

List of Tables

Abstract

Microprocessor-based instruments are traditionally more difficult to design and operate than their analog counterparts. This thesis details a structured approach to an operating system suitable for stand-alone use in an instrument. The system is demonstrated in an automated instrument used to determine the mechanical properties of steel.

The proposed I/O system allows a computer program to converse with the operator using a "natural" language. The system functions as a high level interface between the application programmer and the computer user. This sub-system uses the structured design method to define a powerful virtual operating system that can be placed in ROM. The aim of the operating system is to simplify the application program interface with the hardware of the computer system. The system is interesting in that the virtual devices are defined in independent levels so that additional system capabilities may be added at a later date. The system is designed to be "Fail-Proof" in that erroneous user entries cannot cause the system to halt or abort. This aspect is very important in instrumentation automation because the user is certain to enter wrong

data on occasion.

The structured approach to system design is based upon the construction of new components out of previously defined ones. The distinction between software and hardware becomes immaterial because this approach is based upon high level functions rather than physical implementation. Groups of functions define packages which characterize the set of operations which may be performed by the system.

1. INTRODUCTION

The input/output section of a computer system defines the ultimate interface between the HLL (High Level Language) application program and the computer user. Through this sub-system, the user must make his intentions known to the program. Little theoretical research has been done in the area of user-computer interactions because many people consider it to be a minor part of a large programmed system. This thesis details the design and construction of an advanced input/output sub-system for automated instruments. Two parallel problems currently facing computerized instruments are difficulty in design and difficulty in operation. The High level Input/Output (HIO) system proposed simplifies both of these problems.

1.1 Tensile Testing

The motivation behind this research was the development of an automated tensile testing system. A tensile testing system is used to determine the mechanical properties of materials by axially loading a specimen and recording the responses of the sample. The automated tensile testing machine performs intelligent data reduction by recording the raw tensile results and

computing the parameters of the material under test.

1.2 The I/O Problem in Automated Instruments

Automated testing has a problem common to all computerized instrumentation - a large number of operation options. If the machine operator is required to select a value for each option, the instrument will be difficult to use. The problem of massive I/O is overcome in manually operated instruments by having all control parameters stay the same unless changed by the operator.

The I/O system for an instrument should address five problems :

1. Each control value must be alterable by the operator.
2. Most variables will not be changed each time the instrument is used.
3. Each value should have a pre-set (default) value so that a reasonable selection is ensured.
4. The operator usually sets many controls without looking at the control name.
5. The machine should announce the name of the control parameter that the operator is viewing.

1.3 Design Objectives

The input/output system described in this thesis is designed for a low life-cycle cost. In order to satisfy this constraint, some overall objectives are instituted in the system. The main goals of the design are that it be applicable to a large variety of computer based equipment and be simple enough for application programmers to use. From this, sub-goals are established to ensure that the I/O system will be universally used. The I/O sub-system is human engineered in the sense that the interface reacts with the operator (user) without requiring vast HLL programming. The user can change the pace of the computer conversation by entering responses before the queries are put to him.

The I/O system is modularized to operate in levels. Low level routines, (such as serial device drivers), are embedded within higher level routines, (such as string output routines), which are called by the HLL software. This structured programming functions to insulate the HLL program from any un-necessary details about the hardware and operating system characteristics. The structure also permits modification of the operating system by either re-writing old interfaces or building new layers of

software incorporating existing levels.

The I/O section to be described in this section was implemented in PL/M for the INTEL 8080 microprocessor. This language is simple to learn and is easily translated into another structured language such as PASCAL, ADA, or ALGOL. The operating system level structure encourages the programmer to add new levels or change operating characteristics to suit his individual needs. The console I/O system has been re-targeted to the software from the ISIS-80 operating system with reasonable success. Interfacing the I/O system with a more flexible operating system would have resulted in a much smoother user-to-computer communication system.

2. CURRENT COMPUTER I/O SYSTEMS

The chief function of an I/O system is to allow the user to manipulate control parameters of the HLL program. Many different classes of I/O structures are in current use. Two classes are widely used :

- Menu Driven for menu selection — Single character commands
- Line Buffering numerical, strings) — -line oriented inputs(ex:

From the user's point of view, the I/O system is a major part of the operating system. It is the responsibility of the I/O system to maintain an efficient conversation with the user.

2.1 Primitive Systems

I/O subsystems have evolved much since primitive computers were first used. Many different interaction styles have been developed to interface users with particular programs. In many cases, the operating system level boundary is not clear because individual programs control their own I/O.

The interactions of I/O systems with users are based upon models. These models have evolved as users come to expect more services to be provided to them. Early

computer systems had little or no input/output facilities. These systems were designed to test simple algorithms and input was done prior to program execution and output after execution. Input consisted of changing fixed memory locations where data was expected and output was in the form of memory dumps.

Later, the batch environment was invented. In batch systems, the program read all data at the start of execution and printed results in a readable format. The input data was typically prepared off-line and stored on punched cards or paper tape. These systems were advanced compared to earlier systems because the user could make minor modifications to his data and re-run the program. This I/O style endured for many years and is still in wide use in the business data processing field.

The next major advance was the on-line time shared environment. This system allowed the user to monitor program execution and analyze intermediate results. In some cases, on-line error detection mechanisms were incorporated into the user programs to ensure reasonable data was entered. The feedback to the user was immediate when errors were made. Programs on these early on-line systems used much of the code in the I/O section. When

I/O errors were discovered, the system would grind to a halt.

The on-line systems brought with them a disadvantage. With these systems, the user could not simply change one piece of data. When the program was re-run, all of the data to the program had to be re-entered by the user. This problem is severe in the automated testing environment because each run generally uses most of the same data as the previous run. This execution environment left much to be desired from both the programmer's and user's point of view.

Programmer	Too much time was spent writing I/O code.
User	An error would require re-entering all data.

2.2 Contemporary Systems

With the current growth in computer usage, users have come to expect more and more capabilities from the operating system. Computer designers have responded by introducing features in a patchwork fashion because old software had to be compatible with the new operating systems. Many advanced operating systems have abandoned the notion that input and output are the sole responsibility of the user program. These new operating

systems handle most of the data movement up to character manipulation. Even these operating systems do not go far enough in providing support to user programs.

A new model of the input/output process needs to be developed. The operating system must cleanly separate the user from the program desiring the I/O services. The program should only convey its objective to the operating system. The executing program should not be aware of any unnecessary details of data movement such as formatting. The whole idea of the operating system should be the abstraction of data flow. The I/O system designed for the automated tensile testing system will now be described.

The operating system was implemented in a bottom-up fashion. This method allows easy extensions to the system while allowing software self-checks to be incorporated within the code. The lowest level of abstraction is the only part that refers to the actual hardware makeup of the system. At the highest level, even the characters processed by the I/O system need not be known by the user (application) program. Some examples will illustrate the abstraction levels :

```
CI : PROCEDURE () BYTE;
/* WAIT AND READ INPUT CHAR */
  DECLARE  /* PORTS AND MASKS*/
    CSTAT LITERALLY '0EDH' ,
    CDATA LITERALLY '0ECH' ,
    RXRDY LITERALLY '02H' ,
    PMASK LITERALLY '7FH' ;
  DECLARE  /* VARIABLES */
    STATUS BYTE  /* UART STATUS */ ,
    CHAR   BYTE  /* UART DATA */ ;

  STATUS = INPUT(CSTAT) ;
  DO WHILE (STATUS AND RXRDY) <> 0 ;
    STATUS = INPUT(CSTAT) ;
    END ;
  CHAR = INPUT(CDATA) AND PMASK ;
  RETURN CHAR ;
  END CI ;
```

Figure 2-1: Low level I/O routine

```
ADDER : PROCEDURE () ;
/* ADD TWO NUMBERS FROM KEYBOARD */
  DECLARE
    SYSCON LITERALLY '0' ,
    CR     LITERALLY '0DH' ,
   (V1,V2,RESULT) ADDRESS ;

  CALL PROMPT( .('ENTER DATA 1 :' , 0 ) ) ;
  CALL READSI( .V1 ) ;
  CALL PROMPT ( .('ENTER DATA 2 :' , 0 ) ) ;
  CALL READSI( .V2 ) ;
  RESULT = V1 + V2 ;
  CALL WRITESS(SYSCON , .('ANSWER = ' , 0 ) ) ;
  CALL WRITESI ( SYSCON , RESULT ) ;
  CALL WRITESC ( SYSCON , CR ) ;
  END ADDER ;
```

Figure 2-2: High level I/O

Figure 2-1 shows a procedure written at a low level which reads a character from the console. This illustrates the conventional approach to I/O design in that the physical characteristics of the input/output device are reflected in the code to operate the device. This low level of programming restricts the system to operating with only one type of device.

In contrast, the high level procedure shown in figure 2-2 does not refer to any of the characteristics of the physical device. The input device is an abstract object that is dealt with only in terms of formatted data that comes from it. The physical makeup of the device is hidden from the programmer so that another device could be substituted for the console.

2.3 An Effective I/O System

One of the simplest and easiest I/O systems ever devised is the marked knob. Consider the temperature control on a household oven. When the oven is activated, the user decides upon a temperature to operate the oven at. The user can do many things at this point :

1. Change the setting of the control
2. Ignore the control (do not change the value)

3. Look at the current setting

4. Operate the knob outside the proper region (Ie : against the stop)

5. Make a mental note about the oven (Ex : 10 minutes to pre-heat)

The last three options do not determine a setting, so the user must again choose one of the five choices.

This example, although simple, illustrates many inefficiencies in current I/O methods. Simple query I/O systems allow only the first option to be accepted. Menu-driven I/O systems limit the choice to the first two options. These I/O methods are clearly restrictive in the options available to the user. If the users cannot properly interact with a control, the productivity of the operator will be diminished. A case of multiple inputs will clarify the differences between the current approach and the proposed I/O methods.

An oscilloscope is another marked knob I/O system. This differs from an oven controller in that many controls are used. As a further complication, each control will be covered so that examining the current value requires a physical operation. The covers are used to simulate the effect of program variables (data) whose values are hidden. In addition, the user may select an

invalid range and should be warned that it is not possible to accept the input. Also, it is assumed that the user will sequentially operate the controls in a fixed sequence. This last constraint should be examined.

So far, all operations on the controls should appear natural except possibly the ordering of the operations. Some type of ordering is needed to ensure that no control has been overlooked. The ordering constraint appears to be cumbersome when all possible controls on the oscilloscope are considered. The least used controls can be grouped and an access cover placed over these controls. This gives the user another choice in the operation of the instrument - whether or not to open the access cover to the control group. Although this sounds strange, this method is actually used in modern oscilloscopes. For example, a screwdriver is needed to focus the instrument and the calibration controls may be adjusted only with the instrument open for servicing.

Nested control grouping can be continued at many levels. For example, an assembly line worker need not have access to the controls because they need never be varied. Control hiding is important so that the user's concentration is not drawn from the important controls.

The I/O system must address two related problems :

1. The questions are likely to be repetitious and are soon ignored by the instrument operator

2. The operators have varying degrees of expertise with the instrument

The first problem is minimized by control group covering and the second is eliminated through user type-ahead. Type-ahead is when the user responds to the input request before it is actually issued. Very few intelligent instruments offer type-ahead because the keyboard input is not buffered (edited).

Another requirement of the I/O system (and the entire instrument in general) is that some means of resetting the instrument is needed in case a drastic operation error is made. The system re-start must be orderly so that a minimum amount of typing is required to pick up from the point of interruption.

The separation of the user from the application program results in many benefits.

- Application programmers can concentrate on computation and control and leave I/O to a separate system.

- The I/O interaction is uniform. Any feature supported by the operating system is available on any I/O call.

- The I/O system is portable. It is written using a simple-to-understand programming.

3. PROPOSED I/O SYSTEM

The proposed input/output system incorporates both the oven and oscilloscope models within it. The goal is to make the I/O process simple for the application program, yet flexible for the computer operator. Seen by the program, the I/O system is no more complex than a vastly simpler one. The user, however, sees a computer system that behaves in an almost human fashion. Furthermore, the detection of input errors in an orderly fashion is ensured by checking all user input at the appropriate level.

3.1 System Goals

The following goals were set for the operating system :

- User goals :

    * Free format input (no fixed columns).
    * Allow all five options discussed in the oven example.
    * Request input (prompt) only when the user makes a mistake or stops supplying responses to questions.
    * Allow input line editing (backspace, delete line, echo line).
    * Allow comments to be freely interspersed

with data.

* Allow 'special' characters to be entered (Ex: debugger and baud change).

- Application program goals :

* Code is to be self-initializing if possible.

* Only three parameters to be passed to the operating system : prompt message, read type, and the symbolic name of data to be read.

* The operating system should handle all input errors.

* Protect (hide) as much of the operating system as possible from the application program (Note : only one simple variable is shared between the application program and the operating system).

* Allow the application program to instruct the I/O system to flush old information.

- Hardware goals :

* Only two I/O routines exist : CI and CO for console I/O.

* No formatting is done by the hardware.

- Modifiability goals :

* Structured design to be used ; operating system levels must be cleanly separated.

* The system cannot use special cases to fix

an apparent quirk in the software. Bugs in the system are to be fixed at their sources rather than where the error is encountered.

* Special functions of the system are to be centralized. This ensures that changes to the functions are correctly incorporated into the operating system by any routine that uses that function.

* Allow the system to be easily upgraded.

- Re-targetability goals :

* Allow easy modification for similar hardware (Ex: SBC 80/10 board).

* Allow easy modification for similar software (Ex: CP/M or ISIS operating system).

* Allow easy modification for different processors.

3.2 Structure of the I/O System

The modular approach was used to implement the I/O system. Two different design strategies were used : Top-down and Bottom-up. The Top-down method is most useful when the highest level structures are fixed at one time. The Bottom-up method is better suited when the lowest level structure is specified in advance. Both strategies were used in this project in what I call the "Middle-out" stategy.

In the Middle-out approach, an intermediate goal is established. The intermediate goal in the case of the I/O system is that all data transfers proceed one byte at a time with no "look-ahead" or "check-behind". This goal was selected because of the success of the PASCAL run-time system [WIRTH 75, GROGANO 78] which allows logical device I/O. The design now proceeds Top-down from the highest level to the intermediate level and Bottom-up from the lowest to the intermediate level. At some point, both levels will meet and be fully compatible.

This method has the advantage of partitioning responsibilities for the operating system development into two groups. During the coding phase, the groups should require as little communication as possible. The designers should agree first on their respective coding strategies. For example, each procedure should accomplish one generalized function rather than grouping several functions into one procedure. If possible, the procedures should verify the input parameters for validity.

The I/O system was inspired by TROLL, a very large software system. All input to this program is in free

format and input is requested only when the user types none in. The TROLL system is designed as a fully integrated econometric modelling system with thousands of procedures within it. The TROLL language has so many options that a new I/O system was developed to deal with all of the default values that would generally be used in a particular execution of the program. TROLL was one of the first generation of intelligent systems where commands and responses were differentiated by the operating system.

TROLL's major advance over existing I/O systems was that the commands given to it were generally incomplete. High level operating systems allow the user to direct the flow of information without the application programs being involved in any aspect of the actual data movement. The more responsibility that the operating system possesses, the less work needs to be done in application programming. In TROLL, this reduction was needed because thousands of I/O calls are made throughout the program.

3.3 Components of the HIO System

The I/O system is structured into six hierarchical levels. At each level, the hardware appears more and more flexible (or conversely, the software becomes more

flexible). This level structure allows software validity assertions to be made in many places. The I/O system levels are :

1. Physical I/O : Handshake control
2. Logical I/O : Output to one of three devices
3. Buffered I/O : Input line editing, output character mapping
4. Typed simple I/O : Integer, real, string, and logical data typed
5. Interactive I/O : Prompting, query, and ignore operations
6. Formatted output : Fixed format for tables (real, string, and integer)

Since level three serves as the intermediate goal in this "Middle-out" design, all software support at lower levels is invisible to the application program. For example, the console keyboard cannot be polled for ready status by level 4-6 software because all physical aspects of the console input are hidden. Level six has been implemented in another module (WRITEST) to show how to expand the I/O system to application specific routines.

3.4 I/O System Software Levels

The HIO system is structured into layers (or levels). The layer structure uses the concept of virtual

hardware, which makes software appear to operate like hardware. Once this is done, the hardware and software functions become indistinguishable.

The advantage of the virtual hardware approach is that the system can be specified in functional terms without referind to its actual implementation. The system emulates all hardware that does not physically exist. As an example of virtual hardware, consider the intelligent terminal. This may be used by the I/O system software regardless of whether it is real or merely emulated by the software.

The I/O system levels describe virtual hardware to be used by higher levels of the system. The following paragraphs describe the I/O system levels in detail.

Level_1

These routines are the hardware drivers for the I/O devices. The software waits for the device to be ready before attempting to read or write data. The operating system uses three input drivers and three output drivers. The three input drivers interface the system console keyboard and printer to the operating system. The output drivers interface the system display device and printer

to the operating system. Table 3-1 describes all hardware drivers used by the operating system. Note that VPO is treated by the I/O system as if it were a hardware driver although it is really a software driver. This is an example of the type abstract reference which this system permits.

Device Name	Functional Description
CI	Console input - read from keyboard
CO	Console output - write to system display device
PI	Printer input - read serial data from printer
PO	Printer output - write data to be printed
PSTAT	Printer status - read status of printer port
PCOM	Printer command - write commands to printer port
VPO	Virtual printer output - write to SAVED queue

Table 3-1: Level 1 Devices : Hardware

Level_2

These routines select among the level one hardware and software drivers for the character output function. Table 3-2 lists the logical output devices.

Output character editing is done to make the output devices look like smart terminals. In this case, the logical output device automatically generates line feed characters when carriage returns are printed. The output device gains intelligence through software routines at this level.

Device Number	Device Name	Functional Description
0	SYSCON	System console output
1	SYSPRT	System virtual printer
2	SYSBUF	System buffer (for edited output)

Table 3-2: Level 2 Devices : Logical

Level_3

This level completes the intermediate I/O system level. Three processes are included at this level :

1. Input line editing.
2. Buffered character read (no line available).
3. Output character conversion.

The principal aim of this level is to simplify HLL programming of the I/O, yet leave enough flexibility to allow for interactive I/O.

The input line editing task groups the console input characters into desired typed-in lines. Table 3-3 list the characters which have special meanings to the line editor.

Character	Meaning
^B	Switch baud rate - instructions given to user
^G (BEL)	Enter/exit from intra-line comment
^H or DEL	Delete last character
<CR>	End of line, finish editing
^R	Echo current line (hardcopy edit only)
^U or ^X	Delete whole line
<ESC>	Enter debugger

Table 3-3: Line Editing Characters

The operating system protects itself from user errors by warning the user when backspacing at the line beginning or typing more than 80 characters on a line. The warning indicator is the terminal bell.

The buffered character read process copies the next character in the input buffer to a variable named CHAR. This routine serves to minimize HLL I/O interaction with the operating system by prohibiting access to the actual line buffer. The I/O system makes all character reads

appear as if the reading is done directly from the keyboard. Built-in protection ensures that a character read issued when the line buffer is empty will be marked as an error. Note that this type of error is only caused by programming flaws in high level I/O functions. The terminal bell is rung if a read is attempted while at the end of the line.

The basic console I/O data ports have been extended by level 3 to intelligent terminal ports. The intelligent terminal allows off-line character and line editing to be performed without support of the rest of the I/O system. In addition, the console now acts as a software front panel with the <CTRL B> and <ESC> input keys. Information hiding at lower levels ensures that changes to higher levels of software will not affect the operation of this smart terminal. The lack of interference also means that low level modifications to the system will not have a side-effect on any program using level 3. software.

Level_1

This section of the I/O system interfaces the (intelligent) console device to the hardware/software

architecture. This level converts data values from their internal representation (binary bytes) to their external representation (ASCII characters) and vice-versa. Procedures are included for I/O of character strings, integers, and real numbers. Another procedure is available to read logical values from the console in the for of "YES" and "NO" answers. This level gives the higher level structures the illusion of using a highly sophisticated system console. In addition to the level 3 functions, this console can perform implicit conversions to and from internal binary formats.

The binary typed data formats are defined by the underlying programming standards of the system. In this case, the language PL/M defines the data types. The string data type, however, is stored in a different format from the standard. Strings are stored in memory with a zero byte terminating the string rather than using string length counts throughout the HLL program.

The ASCII inputs required for level 4 read operations are defined in table 3-4.

Level_5

Level 5 structures introduce the concept of the

Integer	series of digits, no sign allowed.
Real	<sign><integer>.<integer>[E<sign><integer>]
String	Series of characters up to <CR> including the usual separators.
Logical	'Y' or 'N' followed by any characters up to the item separator.

Table 3-4: ASCII Input Types

interactive terminal. By interactive, it is meant that the dialogue between the computer and user is dependent upon the user (no fixed dialogue is used). Unlike level 4 routines, which operate on data, this level operates upon the structure of the entered data. That is, the elements which separate the input data values.

This level implicitly defines the input line structure to be a series of entries. Each entry is of the form :

<Input value><Separator>

The separator is defined below as the EOV indicator (character). If a separator appears before a <CR>

character, it is ignored. This allows the two following input lines to be treated similarly :

1,2,<CR>

1,2<CR>

Several special characters are used at this level :

?	Query old (current) value of the variable
,	Marks the end of a value (EOV)
<CR>	Marks the end of an input line (EOLN) as well as EOV

Certain rules are followed for unambiguous entries :

1. Check for "?"
2. Try to read data item (stopping before or at EOV)
3. Verify EOV found, if not then error
4. Skip over EOV character unless it is also the EOLN

Before the user program attempts to read a data item, the PROMPT processor is called with the prompting message. If the user has not already responded to the pending read request, the prompt is issued. The PROMPT processor prints the query and requests a new input data line only if the input character is the EOLN. This

allows the HLL program to always query the user without worrying about cluttering the I/O dialogue.

Level_6

Level 6 output demonstrates an application oriented level of output. This level is implemented in a separate module from the rest of the I/O system to show that it is truly independent of the rest of the I/O system. This layer provides fixed format output of integers, reals, and string types. Real numbers may be scaled before outputting using the special routines described in level 6. The system buffer (device 2) is used to accumulate data before it is sent to the desired output device.

3.5 I/O System Commands

The input/output system allows very limited access to it by user program. The reason for this is two-fold :

1. Users may use high level code for readability
2. A small I/O set can be easily designed for maximal error detection and correction.

As a side effect, hierarchical design produces produces programs which either fail the first time or always work.

This section will discuss I/O commands implemented

on levels 4, 5, and 6. Three output devices are defined in table 3-2 along with their mnemonics. The device SYSBUF consists of a buffer array and an associated buffer index. The index has the range [0 .. BufferSize-1] where BufferSize is the length of the array. Writes to the SYSBUF device store characters at

```
OBUF(OBUFSIND)
```

memory location and then increment OBUFSIND. The device is used as follows :

```
OBUFSIND = 0 ;
/* write to SYSBUF device */
/* output OBUF buffer to the desired device */
```

Four sets of I/O commands are used by the HLL software :

1. Initialize : Flush the input buffer (Level 3)
2. Prompt/Read : Read a data value (Level 4)
3. Unformatted output : Cannot predict output length (Level 5)
4. Formatted output : Can edit level 5 outputs (Level 6)

The initialization command is simply

```
CHAR = CR
```

which forces the I/O system to display the next prompt message. This is used whenever the HLL programmer is unsure about the state of the input buffer and wishes to clear it.

The Prompt/Read set of operations consists of a prompt followed by a read request. The demonstration program described later gives examples of the Prompt/Read commands to read various data types (routines PROMPT, READSI, READSR, READSS, and READSL). The program also shows how the unformatted output is used in routines WRITESI, WRITESR, and WRITESS. The formatted output routines write to the system buffer and then copy parts of the buffer to the desired output device. Numerous examples of formatted output are shown in the TSTR modules FILLSIN-FINAL-DATA and the main program module (TSTR).

4. RESULTS

In order to demonstrate the HIO system operation, it has been modified to run with a development system. The disk operating system used has several unfortunate characteristics which preclude using an actual print example. These problems concern internal line editing done by the development system prior to actual character I/O. The following examples have therefore been re-produced from running tests on the development system.

4.1 Demonstration of the HIO system

Using a modified version of the hardware drivers, the I/O system was re-targeted to the INTEL ISIS-II disk operating system. A demonstration program was prepared to illustrate the system flexibility. The demonstration program runs in 10 kilobytes of user code and 0.5 K of user variables layered upon 14 K of the disk operating system. Of this 10 K, 6 K is used by the I/O system, 1 K by the arithmetic processor(MATHUT), and 3 K by the floating point subroutine package (FPAL).

Two levels of editing have been described above : level 3 and level 5. Level 3 editing characters form the immediate editing characters which operate as soon as they are typed in. Level 5 edit characters operate only

when the application program issues a read request to the operating system.

4.2 Line Editing Characters

The first character to be discussed is the comment entry/exit indicator. For example, if the user types '^G,x,x,^G' only the 'xx' is transmitted to the level 4 software. The screen displays the following characters :

' (* xx *) '

The usage of the level 3 (line editing) characters will be described by example. The user wishes to type the number '1234' into the input line. Comments are used to indicate what the user would be doing with each edit.

The usage of the monitor entry and baud rate switch are implementation dependent and therefore will not be discussed.

4.3 Value Editing Characters

The level 5 (value) editing characters will be exemplified as with the level 3 functions. The application program in this example repeatedly requests four inputs - integer, real, logical, and string. The

```
1. 1235  (* NOW DELETE 5 (BY USING ^H) AND
   REPLACE WITH 4 *)  \54

2. 1111  (* DELETE THE WHOLE LINE BY
   USING ^U *)   <XXX>

3. 1235  (* EDIT 5 AS BEFORE AND ENTER
   ^R TO RE-TYPE LINE *)  \54 <XXX>
   1234
```

Figure 4-1: Character Editing Sample Output

user program issues appropriate prompts before each read and begins with the integer read. Three types of output are written to the user's console : application program prompts, HIO system messages, and user inputs. To distinguish these three entries in figure 4-2 below, application program prompts are written in upper/lower case, HIO messages are in upper case, and user entries are in lower case.

The demonstration program that was used for this example is listed in figure 4-3.

```
Enter Count : wrong anser is entered
ERROR IN INTEGER, INVALID CHARACTER *w* AT COLUMN 1
Enter Count : 17,1.01  (* answer next
                          question before asked *)
Use Plan A ? (* query the old value *) ?
YES>Use Plan A ? (* let the value default.  note that any
                    character can appear in a comment.
                    since nothing but a carriage return
                    will be typed, the HIO system
                    indicates this *)   <CR>
Customer Name ? ,,,  (* skip to 'plan a' question
            *)  n  (*   answer it with 'no'
            *)  ,  (*   skip to count question
            *)    (* re-type to check the
                     resulting line *)   <XXX>
,,,n,  (* that looks ok, now enter that line *)
Enter Count :  (* end of demonstration *)
```

Figure 4-2: Value Editing Program Example

```
DECLARE COUNT BYTE ;
DECLARE RCONST REAL ;
DECLARE PLANA BYTE ;
DECLARE CUST (20) BYTE ;
DECLARE CHAR BYTE EXTERNAL  /* FROM OP-SYS */

CHAR = CR ;  /* FLUSH THE READ BUFFER
                BEFORE STARTING */

START :  /* START OF READ LOOP */
CALL PROMPT( .('Enter Count :',0));
CALL READSI( .COUNT ) ;
CALL PROMPT( .('Enter Scale Factor',0));
CALL READSR( .RCONST );
CALL PROMPT( .'Use Plan A',0 ));
CALL READSL( .PLANA );
CALL PROMPT( .('Enter Customer Name',0));
CALL READSS( .CUST, LENGTH(CUST) );

GOTO START ;
END ;
```

Figure 4-3: Demonstration Source Program

5. CONCLUSIONS

This thesis shows how software for microcomputers can be treated as an effective interface between the user and the application programmer. This view is facilitated by the construction of "virtual hardware" which emulates real hardware. Virtual hardware is supported through the hierarchal structure of the architecture.

The HIO system is shown to efficiently map the user's objectives into keyboard entries. The measure of efficiency in this case is the reduction in keystrokes for an error prone user.

The design of the system permits its use in many microprocessor based instruments. The system is easy to use for both the instrument designer and the user. The level structure permits system programmers to extend the HIO system to suit individual needs such as high level output devices.

The HIO system was installed in an automated tensile testing system and produced spectacular results. An operator with no previous computer experience learned to use the I/O system within a few minutes. The user then quickly lost the natural apprehensive feeling against computers. This is attributable to the conversational

nature of this I/O system compared with the question and answer approach of other I/O systems.

In conclusion, the HIO system shows that high quality software can be developed for a microprocessor operating system. This software is either very simple or very complex depending on the point of view.

I. Source Code for the HIO System

The HIO system has been implemented in the standard language PL/M. The procedures are broken down into three parts by function. The HIO system is currently targeted to run on an INTEL 80/20-4 CPU board and uses approximately 500 bytes of RAM.

The source code is available from Lehigh University in both CP/M and ISIS formats from the Electrical and Computer Engineering Department. The three module available are UTILS.ASM, IOUTIL.PLM, and WRITFT.PLM. The module UTILS.ASM is written in assembly language for the 8080 microprocessor and contains the hardware drivers for the I/O devices (level 1). IOUTIL.PLM contains levels 2-6 of the HIO system. WRITFT.PLM contains the code for level 6 of the I/O system. The languages used for the HIO system allow it to be easily re-targeted to another system or another processor.

In order to use the HIO system, it must be interfaced with a floating point I/O processing package. The modules needed are included with the HIO system and are called LAS2FL and LFL2AS. The first module is the ASCII to floating point conversion routine and the second is the floating point to ASCII routine. In order to use

these modules, a floating point package must be used. The real arithmetic for this application was done with the INTEL FPAL library which emulates a hardware floating point processor.

To sum up, the following modules are needed to implement the HIO system on and 80/20-4 CPU board.

UTILS.ASM	Level 1 of the HIO system.
IOUTIL.PLM	Levels 2-6 of the HIO system.
WRITET.PLM	Level 6 of the HIO system.
LFL2AS.PLM	Floating point to ASCII conversion.
LAS2FL.PLM	ASCII to floating point conversion.
FPAL.LIB	Floating point arithmetic library.

All of the above module are available from Lehigh University except the FPAL.LIB module which is licensed only through INTEL.

Vita

Bruce A. Muschlitz, son of James and Barbara, was born in Bethlehem, Pa in 1958. He received the B.S. degree in Computer Engineering from Lehigh University in January, 1979. Currently, he is working toward the M.S. degree in Electrical Engineering at Lehigh University. While persuing the M.S. degree, he worked for two years as a teaching assistant in the Electrical and Computer Engineering department in various undergraduate courses. In addition, he was employed as a research engineer at Carpenter Technology Corporation for six months.

Mr. Muschlitz is a member of both the IEEE and the IEEE Computer Society.

Bibliography

[GLASS 79] R.L.Glass.
Software Reliability Handbook.
Prentice-Hall, 1979, chapter 3.3.

[GROGANO 78] P.Grogano.
Programming in PASCAL.
Addison-Wesley, 1978.

[KERNIGHAN 81] B.W. Kernighan and J.R. Mashey.
The UNIX Programming Environment.
COMPUTER 14(4):12-22, April, 1981.
IEEE Journal.

[KNUTH 68] D.E.Knuth.
The Art of Computer Programming.
Addison-Wesley, 1968, chapter 2.2.1 and 2.2.2.
Knuth Discusses CIO queues briefly.

[MAKSOUDIAN 69]
A.L.Maksoudian.
Probability and Statistics.
International Textbook, 1969, pages 272-275.

[SHAW 74] A.C.Shaw.
The Logical Design of Operating Systems.
Prentice-Hall, 1974.

[WIRTH 75] K.Jensen and N. Wirth.
PASCAL User's Manual and Report.
Springer-Verlag, 1975.
Niklous Wirth is the author of PASCAL.