1973

Design and complete definition of a systems programming language for data communications

Eldon Jerome Niebaum

Iowa State University

Follow this and additional works at: https://lib.dr.iastate.edu/rtd

Part of the Computer Sciences Commons

Recommended Citation

https://lib.dr.iastate.edu/rtd/4962

This Dissertation is brought to you for free and open access by the Iowa State University Capstones, Theses and Dissertations at Iowa State University Digital Repository. It has been accepted for inclusion in Retrospective Theses and Dissertations by an authorized administrator of Iowa State University Digital Repository. For more information, please contact digirep@iastate.edu.
INFORMATION TO USERS

This material was produced from a microfilm copy of the original document. While the most advanced technological means to photograph and reproduce this document have been used, the quality is heavily dependent upon the quality of the original submitted.

The following explanation of techniques is provided to help you understand markings or patterns which may appear on this reproduction.

1. The sign or "target" for pages apparently lacking from the document photographed is "Missing Page(s)". If it was possible to obtain the missing page(s) or section, they are spliced into the film along with adjacent pages. This may have necessitated cutting thru an image and duplicating adjacent pages to insure you complete continuity.

2. When an image on the film is obliterated with a large round black mark, it is an indication that the photographer suspected that the copy may have moved during exposure and thus cause a blurred image. You will find a good image of the page in the adjacent frame.

3. When a map, drawing or chart, etc., was part of the material being photographed the photographer followed a definite method in "sectioning" the material. It is customary to begin photoing at the upper left hand corner of a large sheet and to continue photoing from left to right in equal sections with a small overlap. If necessary, sectioning is continued again — beginning below the first row and continuing on until complete.

4. The majority of users indicate that the textual content is of greatest value, however, a somewhat higher quality reproduction could be made from "photographs" if essential to the understanding of the dissertation. Silver prints of "photographs" may be ordered at additional charge by writing the Order Department, giving the catalog number, title, author and specific pages you wish reproduced.

5. PLEASE NOTE: Some pages may have indistinct print. Filmed as received.

Xerox University Microfilms
300 North Zeeb Road
Ann Arbor, Michigan 48106
NIEBAUM, Eldon Jerome, 1939-
DESIGN AND COMPLETE DEFINITION OF A SYSTEMS
PROGRAMMING LANGUAGE FOR DATA COMMUNICATIONS.

Iowa State University, Ph.D., 1973
Computer Science

University Microfilms, A XEROX Company, Ann Arbor, Michigan

THIS DISSERTATION HAS BEEN MICROFILMED EXACTLY AS RECEIVED.
Design and complete definition of
a systems programming language for data communications

by

Eldon Jerome Niebaum

A Dissertation Submitted to the
Graduate Faculty in Partial Fulfillment of
The Requirements for the Degree of
DOCTOR OF PHILOSOPHY

Major: Computer Science

Approved:

Signature was redacted for privacy.

In Charge of Major Work

Signature was redacted for privacy.

For the Major Department

Signature was redacted for privacy.

For the Graduate College

Iowa State University
Ames, Iowa

1973
TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>CHAPTER I.</th>
<th>INTRODUCTION</th>
<th>page</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAPTER II.</td>
<td>FUNCTIONS OF COMMUNICATIONS PROCESSORS</td>
<td>4</td>
</tr>
<tr>
<td>CHAPTER III.</td>
<td>ELEMENTS OF SYSTEMS PROGRAMMING LANGUAGES</td>
<td>13</td>
</tr>
<tr>
<td>CHAPTER IV.</td>
<td>A COMMUNICATIONS ORIENTED SYSTEMS PROGRAMMING LANGUAGE</td>
<td>20</td>
</tr>
<tr>
<td>CHAPTER V.</td>
<td>FORMAL DESCRIPTION OF THE LANGUAGE SYNTAX</td>
<td>35</td>
</tr>
<tr>
<td>CHAPTER VI.</td>
<td>FORMAL DESCRIPTION OF THE LANGUAGE SEMANTICS</td>
<td>41</td>
</tr>
<tr>
<td>CHAPTER VII.</td>
<td>A SAMPLE DATA COMMUNICATIONS ENVIRONMENT</td>
<td>70</td>
</tr>
<tr>
<td>CHAPTER VIII.</td>
<td>SUMMARY AND CONCLUSIONS</td>
<td>74</td>
</tr>
<tr>
<td>BIBLIOGRAPHY</td>
<td></td>
<td>77</td>
</tr>
<tr>
<td>ACKNOWLEDGMENTS</td>
<td></td>
<td>80</td>
</tr>
<tr>
<td>APPENDIX A.</td>
<td>THE SYNTAX OF INTERCOM IN META PI</td>
<td>81</td>
</tr>
<tr>
<td>APPENDIX B.</td>
<td>THE ABSTRACT SYNTAX OF INTERCOM IN VDL</td>
<td>84</td>
</tr>
<tr>
<td>APPENDIX C.</td>
<td>VDL PREDICATES FOR ELEMENTARY OBJECTS IN INTERCOM</td>
<td>87</td>
</tr>
<tr>
<td>APPENDIX D.</td>
<td>AN ABSTRACT INTERPRETER FOR INTERCOM</td>
<td>88</td>
</tr>
<tr>
<td>APPENDIX E.</td>
<td>FLOW OF CONTROL IN A SIMPLE SYSTEM</td>
<td>102</td>
</tr>
<tr>
<td>APPENDIX F.</td>
<td>A SAMPLE PROGRAM IN INTERCOM</td>
<td>104</td>
</tr>
</tbody>
</table>
CHAPTER I.
INTRODUCTION

With the advent of multi-programmed timesharing computers came the potential for remote access to such computers. Whether the need for such access already existed or was created by the technology is not altogether clear. In the decade since multi-terminal remote access became feasible great amounts of industrial and institutional research have been done in the broad area of data communications. Part of that research has resulted in a variety of equipment to interface communications lines to a host computer. These devices have had a wide variety of names such as data adapters, data controllers, communications controllers, and communications processors to name a few. Some of the early controllers were programmable (7), but most were hardwired.

Advances in technology and corresponding decreases in component costs coupled with increased demand for remote access have brought renewed activity in the development and manufacture of programmable communications processors. In one recent article Reagan and Totaro (26) list more than 30 firms now producing or claiming capability to produce such devices. Many of these firms provide software for emulation of existing hardware communications processors. Typically the emulation is for one of the IBM transmission control units. Other firms offer highly modular software routines
from which application systems can be built.

Machine language programming is possible on all programmable communications processors and even micro-programming is possible on some. Assemblers and assembly languages are common vehicles for user creation of a communications processor software system. Carr, Crocker, and Cerf (4) suggest the possibility of a "Network Interface Language (NIL) which would be a network-wide language for writing the front end of interactive subsystems" in the ARPA network. According to Lyle (17) communications handling facilities have been added to ALGOL to form Data Communications ALGOL, but in general higher level languages have thus far not been used to any great extent for communications processing software systems.

The purpose of this research is to isolate those factors which are unique to the processing of serial data communications and to create and describe a programming language in which to design systems for use with programmable processors. Such a language could serve as a vehicle for documentation of existing communications software. Also, it could enable a user to more easily create or modify such software. Additionally, it could free users from dependence on industry supplied routines and encourage use of data communications in application areas heretofore avoided.

It is unrealistic to believe that it is possible or even desirable to produce a programming language with a large num-
ber of unique constructs. Rather the approach taken here is to choose a "minimum" set of constructs from existing programming languages. In this context "minimum" refers to relative simplicity with respect to languages such as PL/1, ALGOL, and AFL. Dijkstra (9) offers some interesting arguments in defense of language simplicity. In brief he argues that a complex language may be beyond the intellectual capacity of most of its users. It can also be argued that a language with few constructs is potentially easier to learn and easier to implement.

The language described herein resembles PL/1 in general structure, while in terms of the operations allowed it bears some similarity to AFL. Further comparisons are made in sections discussing structure of the language. No claim is made that the language is sufficient to describe all communications systems or even sufficient to completely describe any given system. Rather the intent is to provide a tool with which a systems programmer can design and describe the most common and basic communications oriented algorithms.
CHAPTER II.
FUNCTIONS OF COMMUNICATIONS PROCESSORS

Programmable communications processors (abbreviated hereafter as CP) are complex combinations of hardware and software. Their functions can be broadly classified into three groups as shown in Figure 1. The arrows indicate paths which information might take thru the CP system.

Figure 1. Divisions of CP functions.

Townsend (28) gives an extensive list of functions of a CP along with brief descriptions of each. His list, slightly reordered, is given in Tables 1 and 2.

Table 1. Low level functions.

1. Interface Signalling
2. Bit Synchronization
3. Character Synchronization
4. Connection Control
5. Character Assembly/Disassembly
6. Polling/Selection/Contention Resolution
7. Message Assembly
8. Error Detection and Control
9. Code Conversion
10. Data Compression/Expansion
11. Job-Based Routing
12. Terminal Control
13. Network Performance Monitoring/Analysis
14. Terminal Testing
Items 1 thru 4 in Table 1 are primarily functions of CP hardware, while 5 thru 14 may involve both hardware and software. The functions in Table 2 are generally performed by software. A description of some of the functions of a CP are included here to show motivation for some of the language features developed later.

Table 2. High level functions.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Address Based Routing</td>
</tr>
<tr>
<td>2.</td>
<td>Queueing/Dequeueuing</td>
</tr>
<tr>
<td>3.</td>
<td>Priority Scheduling</td>
</tr>
<tr>
<td>4.</td>
<td>Reconfiguration Control</td>
</tr>
<tr>
<td>5.</td>
<td>Fail-Soft Functions</td>
</tr>
<tr>
<td>6.</td>
<td>Time Control/Recording</td>
</tr>
<tr>
<td>7.</td>
<td>Format Control</td>
</tr>
<tr>
<td>8.</td>
<td>Format Conversion</td>
</tr>
<tr>
<td>9.</td>
<td>Communications File Maintenance</td>
</tr>
</tbody>
</table>

Character Assembly/Disassembly

Information presented to a communications processor (CP) from a terminal is normally serial-by-bit. Each bit of a character is received at a specific time relative to the first bit. Transmission can be asynchronous or synchronous \((20,21,29)\). If the transmission mode is asynchronous, then bits of a character are received at specific intervals, but intervals between characters are not regular. Figure 2 represents the message 'HELLO' as it might be transmitted to a CP using asynchronous transmission. The dashes represent inter-character time gaps.
The asynchronous mode requires one bit to indicate start of a character and one or more bits to mark the end of a character. For this reason asynchronous transmission is often referred to as start-stop transmission.

One commonly used computer terminal, the model 33 Teletype, uses an 11 bit code: one start bit (0), seven character bits (ASCII), a parity bit, and two stop bits (1,1). The letter H as it would appear on a transmission line in ASCII-Teletype code is shown in Figure 3.

```
0 0 0 0 1 0 0 1 1 1 1
| |------------------| |--|
start character bits parity stop
bit bit bits
```

Another terminal, the IBM 2741, uses 6 character bits, a parity bit, and only 1 bit each for start and stop. Recently, integrated circuits have become available commercially for converting asynchronous signals from serial to parallel and parallel to serial. These circuits also strip the start and stop bits. Alternatively, for slow speed devices (300 bits/second or less) it is possible to use software routines to assemble and disassemble characters.
Synchronous transmission (6,20,21) is used when it is possible to transmit a message with equal time intervals between characters. This implies that the terminal must be capable of buffering an entire message before transmission. Message transmission is always preceded by transmission of a series of synchronization (SYN) characters which enable the receiving station to establish timing for the message bits which follow. Two or more SYN characters are normally required to establish synchronization. Since start and stop bits are not used, fewer bits are needed for message transmission. Timing is most often provided by the modems (modulator/demodulator) which also shape the signal for telephone line transmission.

Polling / Selection / Contention Resolution

A CP system issues an invitation to a terminal to transmit information to the host computer. In simplest form this invitation may merely be recognition of any incoming information. More complicated schemes involve transmission of a control message to specific terminals in a sequence as an invitation (polling). This scheme is often used when multiple computer terminals share a transmission line. Selection is the process of choosing a specific terminal to receive output from the host computer. When two or more devices bid for the same resource a contention situation exists. The CP can resolve such conflicts by choosing one device and
postponing requests from the other devices.

Message Assembly

Information input to a CP may be character buffered or message buffered. Buffer storage can be in memory in the CP or in a memory shared with an adjacent processor. The IBM 2702 for example assembles characters and stores them in the attached computer memory in conjunction with the multiplexor channel on a cycle stealing basis.

A message can be fixed length or variable length with the latter being more common. Variable length messages require the CP to examine the input stream for an end of message character or sequence of characters. Messages from asynchronous terminals usually terminate with RETURN or X-OFF. The commonly used Binary Synchronous Communications (BSC) scheme (13) frames a message between start characters (STX or SOH) and ending characters (ETX or ETB). Other control characters (ACK, NAK) can act as a complete control message. Fixed length messages are less flexible and require that the terminal itself be buffered.

Some line interfaces to a CP have a single register for the dual tasks of assembling and buffering characters. This structure implies that after assembly, the register contents must be accessed within 1 bit time or data overrun can occur and the data lost. Asynchronous terminals transmitting at 110 baud (bits per second) will supply a new bit each 9
milliseconds. The DEC DC-11 interface (8) is an example of this type of interface and requires that the data buffer be read within 18 milliseconds, the time for two stop bits. A line interface may be double buffered so that following assembly the assembled character is stored in a holding register. The contents of the holding register must be processed within 1 character time to avoid possible data overrun. An example of this type is the DEC DP-11 interface (8).

Error Detection

Transmission lines are subject to noise signals which can introduce errors in messages (19). For that reason numerous techniques have been developed for detecting and even correcting errors (15,16,20). Communications processors often use an error detection and request retransmission scheme. When an error is detected the processor sends a control signal to the originating terminal requesting transmission retry.

To aid error detection redundant information bits are added to the message. The simplest scheme is to add a parity bit to each character. When odd-parity is used the parity bit is set to 1 if the character has an even number of 1 bits. Even parity is defined in analogous fashion. Parity is sometimes referred to as a vertical redundancy check and enables detection of single bit errors (20). Parity may be used in a horizontal or longitudinal sense for detecting one
bit errors in corresponding bit positions of multiple character messages. In this case an exclusive-or is made of all the characters of the message and the resulting character is transmitted with the message. At the receiving station a similar computation is made and the error characters are compared. This scheme is commonly used with buffered terminals in conjunction with vertical parity.

A more sophisticated error detection scheme based on polynomial division is commonly used in binary synchronous communication (15,16,20). The message is treated as a polynomial with coefficients of 1 in non-zero bit positions. The message polynomial is "divided" by a second polynomial called the generating polynomial with the remainder serving as a cyclic redundancy check (CRC). The CRC enables detection of multiple bit errors, but is quite time consuming if implemented in software (16); therefore, most systems using a CRC have special purpose registers for performing the division.

Code Conversion

Terminals may use a variety of transmission codes, as for example ASCII, EBCDIC, BAUDOT, 6-bit Transcode, etc. with or without parity. Rather than process messages in a variety of codes normally messages are translated to a common code. The code conversion may be done by the communications processor or by the destination processor. Code translation is not
normally implemented in hardware though machine instructions may be designed to aid code translation. An example of this is the IBM/360 Translate and Test instruction (15). One difficulty encountered in translation is that a source character may not have a counterpart in the destination code. One such character is an ASCII Record Separator (RS) which has no counterpart in EBCDIC. A second difficulty arises when an $m$-bit code is translated to an $n$-bit code with $m$ not equal to $n$. In that case many bit combinations have no direct counterparts.

Data Compression / Expansion

Messages for transmission may contain sequences of identical characters. One common sequence is a string of blank characters as a trailing sequence resulting from punched card input where not all columns have been used. Rather than transmit each character in the sequence a CP system may code a replication factor followed by the character. This is an example of data compression. The advantage is that fewer characters need to be transmitted thereby shortening the message transmission time. The receiving terminal may be required to rebuild the original message (data expansion).
Job-based Routing

With hardware controllers a logical connection between terminal and program is virtually the same as the physical connection between terminal, CP, and destination computer. On the other hand programmable CP systems may allow variety in routing under software control. For example terminal to terminal messages can bypass CPU software altogether. In this case the CP acts as a message switcher. Also, messages originating in a host computer can be easily routed to a multiplicity of terminals. One simple routing scheme used by the Houston Automatic Spooling Process (HASP) software includes destination address or identification in a message header.

Function Summary

A CP system processes characters either singly or in sequences. Transformations on characters may be necessary (code conversion) and also on sequences of characters (data compression/expansion). The principal functions of CP system software are input, output, buffering, and transformations on messages. A message in this context is merely an identifiable sequence of characters.
CHAPTER III.
ELEMENTS OF SYSTEMS PROGRAMMING LANGUAGES

The growing complexity of software operating systems has stimulated interest in using high level languages for systems programming. Sammet (27) gives a rather long list of languages which have already been used in the creation of operating systems. With that list the advantages of using high level languages are given and repeated here in slightly modified form in Table 3.

Table 3. advantages of higher level languages.

<table>
<thead>
<tr>
<th></th>
<th>1. Potential for machine independence</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2. Faster coding and development</td>
</tr>
<tr>
<td></td>
<td>3. Easier maintenance and documentation</td>
</tr>
<tr>
<td></td>
<td>4. Greater project managability and segmentation</td>
</tr>
<tr>
<td></td>
<td>5. Simpler system change and redesign</td>
</tr>
</tbody>
</table>

Two characteristics of programming systems given by Sammet are that they do not produce answers as such and they are generally independent of a particular application.

In this chapter features of some existing systems programming languages are discussed and comparisons between them are made. This is by no means an exhaustive discussion of such languages. Rather, the discussion is intended to point out language features which certain authors have considered important and to serve as support for the language developed in later chapters.
Many of the current systems programming languages are dialects of existing languages such as ALGOL (2) and PL/1 (1). One of these, a Language for Systems Development (LSD), has been described by Bergeron, Gannon and van Dam (3). It is a procedure oriented language with a syntax similar to PL/1. A variety of data types is allowed as evidenced by the variable attributes FIXED, FLOAT, CHAR, BIT, POINTER, and AREA. Undeclared variables are flagged as errors and implicit conversions between data types is not performed. The language allows the programmer to "drop directly into assembler language in order to specify exactly what code he wants generated." Assembler language capabilities such as this are not uncommon in systems programming languages. A motivating force for this is that probably the biggest technical problem in using a systems language is execution efficiency. In LSD hardware registers as well as hexadecimal constants, can be used and this is in keeping with the philosophy that "a systems programming language, although high level, should allow the programmer to access all the facilities of the machine."

Storage in LSD can be automatic, i.e. allocated at runtime, or static, i.e. allocated at compile time. However, storage is automatic only in the case of reentrant procedures. Structures, collections of heterogeneous data types, are permitted as in PL/1. In addition to the RETURN state-
ment allowed in PL/1, the construct RETURN TO label; is permitted in LSD thus giving added flexibility in calling procedures.

Levels of multiprogramming are allowed with process synchronization controlled by commands WAIT and POST. LSD extensions to PL/1 include "pointer chasing methods," enhanced character handling facilities, and provisions for debug assistance. Additional operators available are negation (¬), or (|), and ( & ), exclusive or (x), binary left shift (<<), binary right shift (>>), test under mask (TM), and comparison (:).

A language for the IBM/360 computer line called PL360 has been developed by Wirth (32). "The language ... is designed with the aim of providing a clear and systematic exposition of the available computing facilities and a tool which encourages a programmer to write programs in a disciplined, lucid, and readable style while still maintaining control over the optimal use of specific machine characteristics." PL360 is characterized by large numbers of data types such as byte, integer, short integer, logical, real, long real, array, array byte, etc. Manipulation of all the 360 registers is permitted and additional operators have been provided such as "and", "or", "xor", and both logical and arithmetic shift operators. Arithmetic is performed strictly left to right with no operator hierarchy. The lan-
guage generally follows a PL/1 structure with extensions for getting the programmer closer to machine facilities.

According to Wirth "direct programming of input/output operations in PL360 is impractical ...." He suggests that to overcome this shortcoming a "facility (could be) introduced to designate a segment as an interrupt service routine... This approach forces a programmer to make explicit the fact that an interrupt routine is conceptually a closed segment." This approach has been followed in the language developed in subsequent chapters.

A systems programming language called Bliss (34,35) has been developed for a specific computer, the PDP-10. It has basically an ALGOL structure and requires no run-time support facilities. Great variety is allowed in declarations as shown by examples GLOBAL, OWN, LOCAL, REGISTER, FUNCTION, etc. Local values are not preserved on block exit. A distinction is made between a variable name representing a value and a pointer to a value. In particular .X is the value and X is a pointer to the value. In discussing this structure Wulf, et al. (35) remarks that "... it may seem that the notation X and .X for a pointer to the field and value of that field, respectively might better be replaced by @X and X. However, a little comparison will soon show that the dot notation is to be preferred." In brief the authors argue that @X is not a unique inverse of the value of X since there may
be several locations with that value. In spite of this objection the latter notation was adopted in the language developed in later chapters.

In Bliss there are six forms of looping statements including the form

\[ \text{WHILE } \text{exp1} \text{ DO } \text{exp2} \]

which causes repeated execution of exp2 as long as the rightmost bit of the value of exp1 is 1. The authors remark that "all of the loop expressions could be constructed from the \text{WHILE ... DO form."} A structure similar to the \text{WHILE ... DO} is described in succeeding chapters. Relational operators in Bliss yield 1 if the relation is satisfied and 0 otherwise. Assignment is considered to be a dyadic operator.

Two of the most unique features of Bliss are the absence of labels and the absence of a GO TO statement. In place of these features 8 forms of escape mechanism or return mechanism have been provided.

IBM Corporation has a high level language called PL/S for internal use in systems programming, but information relating to its structure is sketchy because it is considered proprietary. The syntax has been described by Wiederhold and Ehrman (31). The language is strongly oriented towards the IBM 360 and 370 lines of computers and it seems to be syntactically close to PL/1. Like LSD mentioned earlier this language allows assembler language statements to be
intermixed with the source code. Arithmetic is strictly binary and there are no implicit type conversions.

A subset of PL/1 called EPL was used to develop software for the MULTICS system. Experience with this language has been described by Corbato (5). Strong points of the language proved to be pointer variables, based storage, and dynamic storage. Also, among its strengths were ON conditions and the SIGNAL statement, but these features caused difficulty in implementation. Weaknesses of the language seemed to be "that PL/1 went too far in specifying the exact environment" and "bugs ... caused by mismatched declarations", which might be construed as a language weakness. EPL was chosen for its language richness and modularity in that subsections could be compiled independently. Also, the language was roughly machine independent and it was thought that PL/1 would have wide support.

Hopkins (11) discusses problems associated with using PL/1 itself as a systems programming language. He states that "most of the problems of PL/1 result directly or indirectly from the size and complexity of the language." It is noted also that "PL/1 has a well-defined interrupt system for synchronous interrupts such as fixed point overflow or events that can be transformed into synchronous situations such as end-of-file. However, with respect to asynchronous events such as interval timer overflow or attention button
interrupt, the language is not satisfactory." In the lan-
guage developed in later chapters an attempt has been made to
overcome this weakness.

The languages discussed so far might be classified as
general purpose systems languages in that they are designed
for a large class of systems problems. A second group might
be called special purpose systems programming languages be-
cause they are used primarily for problems of a single class.
One such language is DCALGOL, Data Communications ALGOL, de-
scribed briefly by Lyle (17). This language has been used to
write remote job entry software for the Burroughs B6700
system. Special facilities for handling messages and queues
have been included to form an extended ALGOL. Message buffer
space can be allocated dynamically with the ALLOCATE command
and queues can be manipulated with commands 'INSERT' and
'REMOVE'. Additional information on the language structure
has been unavailable to this author.

In summary selected features of several existing systems
programming languages have been described in brief. The spe-
cific points were chosen to support design decisions in the
language developed in subsequent chapters.
CHAPTER IV.
A COMMUNICATIONS ORIENTED SYSTEMS PROGRAMMING LANGUAGE

In this chapter a systems programming language for data communications is described informally by narrative and example. A formal definition of the language syntax and semantics is developed later. For reference purposes this language will be called INTERCOM.

Features from numerous programming languages have been incorporated in INTERCOM, but it most nearly resembles PL/1 in structure. Perhaps its most distinguishing feature is simplicity. Only integer arithmetic is allowed; the only data types are byte and linear byte arrays; and, while the language gives the appearance of being block structured, in reality it is not. Thus, entry into a "block" does not require saving environment information. In the description which follows the language features are given along with programming examples.

An INTERCOM system consists of a MAIN program and any number of condition blocks used to handle asynchronous interrupts. A system is delimited by the reserved words START and FINISH as shown here:

```
START;
    condition blocks
    main program
FINISH;
```
Each statement begins with a keyword identifier which immediately classifies the statement by type. One advantage of this structure is that no backtracking is required when using a left to right top down parsing algorithm.

Any statement in the body of the program may be labelled. A label is an identifier delimited by left and right matching colons as in :LOOP1: Here the left colon eliminates the need for backup in parsing the program in the same manner as the keyword identifiers mentioned above.

In the sections which follow the structure and functions of the statement types given in Table 4 are described.

Table 4. A summary of statement types in INTERCOM.

<table>
<thead>
<tr>
<th>1. Storage Assignment</th>
<th>5. Input/Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>DECLARE</td>
<td>INPUT</td>
</tr>
<tr>
<td>ALLOCATE</td>
<td>OUTPUT</td>
</tr>
<tr>
<td>FREE</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>2. Interrupt Control</th>
<th>6. Conditional</th>
</tr>
</thead>
<tbody>
<tr>
<td>ON INTERRUPT</td>
<td>IF-THEN-ELSE</td>
</tr>
<tr>
<td>ENABLE</td>
<td></td>
</tr>
<tr>
<td>DISABLE</td>
<td></td>
</tr>
<tr>
<td>SIGNAL</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>3. Transfer</th>
<th>7. Looping</th>
</tr>
</thead>
<tbody>
<tr>
<td>GO TO</td>
<td>WHILE-block</td>
</tr>
<tr>
<td>CALL</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>4. Assignment</th>
<th>8. Control</th>
</tr>
</thead>
<tbody>
<tr>
<td>LET</td>
<td>WAIT</td>
</tr>
<tr>
<td></td>
<td>HALT</td>
</tr>
<tr>
<td></td>
<td>RETURN</td>
</tr>
</tbody>
</table>
Table 5. Key symbols other than statement identifiers.

| 1. System Delimiters       | 5. Storage Types        |
|                           | BYTE                    |
| SYSTEM; - FINISH;         | BUFFER                  |
| 2. Block Delimiters       | HIERARCHY               |
| BEGIN; - END;             |                         |
| 3. Label Delimiters       | 6. Device               |
| :label:                  | UNIT                    |
| 4. Comment Delimiters     | 7. Functions            |
| "-----"                  | CHAR                    |
|                           | HEX                     |

Storage Assignment

All variables must be declared and the declaration is global to the entire program. Each name entry in a declaration may be followed by attributes to be held by the variable being declared. These attributes may be BYTE, HIERARCHY, or BUFFER (or any combination of the three). A byte is normally considered to be 8 contiguous bits addressable as one unit. The concepts described are equally valid for N bit bytes. The BYTE attribute must be qualified by specifying the number of bytes to be used for the variable. For example

 DECLARE A BYTE(3);

indicates that 3 contiguous bytes will be reserved and can be referred to by the variable A.

Arrays can also be specified as, as for example

 DECLARE B(5) BYTE(2);

which specifies 5 groups of two bytes each and designated by B(0), B(1), ..., B(4). All arrays are zero origin.
Most computer systems have a hierarchy of storage devices ranked according to speed. The HIERARCHY attribute allows the systems programmer to specify the level of storage to be used. These levels are installation dependent, but will generally follow the structure given in Table 6.

Table 6. A hierarchy of storage.

<table>
<thead>
<tr>
<th>level</th>
<th>storage device</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>register (fastest)</td>
</tr>
<tr>
<td>1</td>
<td>main memory</td>
</tr>
<tr>
<td>2</td>
<td>bulk memory</td>
</tr>
<tr>
<td>3</td>
<td>drum</td>
</tr>
<tr>
<td>4</td>
<td>disk (slowest)</td>
</tr>
</tbody>
</table>

As an example of the HIERARCHY attribute consider the declaration of a variable, R1, as a two byte register,

```
DECLARE R1 BYTE(2) HIERARCHY(0);
```

or the declaration of a single byte of main memory,

```
DECLARE FLAG HIERARCHY(1);
```

If the HIERARCHY attribute is not used then level 1 is assumed. The amount of storage available at each level is installation dependent.

The BUFFER attribute provides for specification of buffer areas for unsolicited input. For example a terminal may begin transmission to a computer system without first receiving an invitation to transmit. The input is stored in the area specified by the BUFFER attribute. The buffer area must be further qualified by a UNIT specification to associate the
buffer with a specific device. Units are specified numerically and are installation dependent. Consider the declaration

```
DECLARE BUF1(72) HIERARCHY(1) BUFFER UNIT(3);
```

which establishes a 72 byte buffer area for unsolicited input from the device designated as unit 3. Some devices such as typewriter terminals may require only a single byte input buffer.

Storage reserved by DECLARE statements is statically assigned at compilation time. In addition storage may be dynamically assigned and released from storage during program execution. Commands for using dynamic storage are ALLOCATE and FREE. Storage attributes are the same as for DECLARE. Dynamic storage makes storage reusable at the expense of execution speed. In the example shown here

```
ALLOCATE BUF1(72) HIERARCHY(1) BUFFER UNIT(3);
```...

```
FREE BUF1(72);
```

a buffer for asynchronous input is allocated and later in the program that space is made reusable.

### Interrupt Control

The handling of asynchronous interrupts is facilitated by ON INTERRUPT blocks which specify actions to be taken in the event of specified interrupt types. The statement

```
ON INTERRUPT(2) UNIT(3) RETURN;
```
specifies that a type 2 interrupt on unit 3 requires no action other than to return from the interrupt. If action is required then a BLOCK of statements can be used in place of RETURN. A label is required on the ON INTERRUPT statement.

The ON INTERRUPT statements are similar to ON conditions in PL/1. They differ in that most PL/1 ON conditions such as stringrange or subscriptrange rely on software testing of conditions internal to the system rather than external interrupts caused by physical devices.

A block is a collection of statements of any type other than DECLARE and ON INTERRUPT and is delimited by keywords BEGIN; ... END;. Blocks are logical groupings and block exit may not imply sequential execution of the next instruction. This type of exit is explained as part of the CALL statement.

The systems programmer can allow or disallow interrupts with the imperative commands ENABLE and DISABLE. Also a specific interrupt can be simulated by the program with the SIGNAL statement which performs the same function as its counterpart in PL/1. Examples of these three statement types are shown below.

```
ENABLE INTR4;
DISABLE CON;
SIGNAL TING;
```
Transfer

Normal sequential execution of instructions can be changed in 4 ways: occurrence of an interrupt, an IF-THEN statement (explained later), and execution of a GO TO or CALL statement. An unconditional branch can be made to any labelled statement by

    GO TO label;

Since all values are global, abnormal entry or exit to blocks needs no special handling.

The CALL statement acts much like GO TO in that normal sequential execution is interrupted and an unconditional transfer is made to a labelled block. The form is

    CALL labelled block;

A RETURN statement within a block returns execution to the next statement following the CALL statement, a structure similar to the GOSUB structure in the BASIC language (22). This simple mechanism allows considerable subroutine capability without adding implementation complexity. Blocks are not reentrant and may not be called recursively.

Assignment

Data movement by assignment is quite flexible in INTERCOM as will be shown. The most common form of assignment is

    LET var = exp;

which has the same format as the BASIC language (22) assign-
ment statement. A simple example would be

\[ \text{LET } X = Y; \]

which causes the single byte designated \( Y \) to be moved to the location designated \( X \). Identifiers can be further qualified by a size factor which indicates the number of consecutive bytes to participate in a given transaction. For example

\[ \text{LET } X.5 = Y(3); \]

means transfer the 5 bytes starting at \( Y(3) \) into consecutive locations beginning with the one designated \( X \). The sequence

\[
\begin{align*}
\text{LET } X &= 4; \\
\text{LET } X(1).9 &= X(0);
\end{align*}
\]

results in 10 elements of an array \( X \) being filled with the value, 4. Recall that arrays are zero origin. The statement

\[ \text{LET } X(0).N = X(1); \]

results in \( N \) bytes of the array \( X \) shifting one position down in the array as if the following sequence had been executed

\[
\begin{align*}
\text{LET } X(0) &= X(1); \\
\text{LET } X(1) &= X(2); \\
\ldots \\
\text{LET } X(N-1) &= X(N);
\end{align*}
\]

The assignment symbol, \( = \), is considered to be an operator and hence can occur in expressions as any other operator. In the statement

\[ \text{LET } X = ((Y=2) + 3); \]

two assignments are made, namely \( Y \) takes the value 2 and \( X \) takes the value 5. The embedded assignment structure is also
used in the APL programming language (12).

Input/Output

Because of installation dependencies input/output statements have sometimes been avoided in defining programming languages. This was the case for example in EULER (33) and ALGOL (2). An assumption is made with INTERCOM that most input will be unsolicited and therefore handled as interrupts. However, an INPUT statement is provided for soliciting input from a specific device. During execution of an INPUT statement no other processing may be done other than interrupt handling. The form for the statement is

\[
\text{INPUT}(3) \; A;
\]

which solicits input from unit 3 to storage designated by A.

Output has the same form as INPUT and is stream oriented as in

\[
\text{OUTPUT}(5) \; B.72;
\]

which outputs 72 bytes to unit 5. There is no formatting of input or output and only a single identifier may be used in the input/output list. In the formal description of the language given in later chapters only the syntax of the input/output statements can be given. The semantics are implementation dependent.
Operations and Expressions

The INTERCOM language has a large number of operators for bit and byte manipulation in addition to those common to most higher level languages. Operators are summarized in the table below.

Table 7. Symbols and operators in INTERCOM.

1. ARITHMETIC
   A+E Addition
   A-B Subtraction
   A*E Multiplication
   A/E Division
   -E Negation

2. LOGICAL
   A&B AND
   A|B or
   A#B Exclusive OR
   $B Complement

3. RELATIONAL
   A<E Less than
   A>B Greater than
   A <= B Less than or equal to
   A >= B Greater than or equal to
   A -= B Not equal to

4. BOOLEAN
   A AND B
   A OR B
   NOT B

5. LITERAL DESIGNATION
   CHAR (STRING)
   HEX (4F2C36)

6. DATA MOVEMENT
   A = B Simple assignment
   A.N = B Multiple assignment

7. ADDRESS
   @B Address of B
   %B Indirect address (address at B)

8. SHIFTING
   << B.N Shift B left N bit positions, zero fill
   >> B.N Shift B right N bit positions, zero fill
Arithmetic is integer on a byte basis unless size factors are specified then the bytes are considered to be ordered right to left as shown

\[ B(N) \ldots B(2) \ B(1) \ B(0) \]

with the least significant bits in byte B(0). For simplicity some implementations of this language may restrict arithmetic to no more than two bytes.

Logical arithmetic is performed on a bit by bit basis so that \( A \& B \) is the logical "AND" of each pair of corresponding bits in bytes A and B. Relational operations produce a numeric value 0 if the relation does not hold and 1 if it does hold. The rightmost bit in a field determines the "truth" value of that field. If that bit is zero the field has a FALSE value and if it is one the field is TRUE. The Boolean operators use only the rightmost bits of the fields. Examples below illustrate these features.

<table>
<thead>
<tr>
<th>expression</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>A=5</td>
<td>5</td>
</tr>
<tr>
<td>B=7</td>
<td>7</td>
</tr>
<tr>
<td>C=4</td>
<td>4</td>
</tr>
<tr>
<td>A&lt;B</td>
<td>1</td>
</tr>
<tr>
<td>A&lt;C</td>
<td>0</td>
</tr>
<tr>
<td>A&amp;B</td>
<td>1</td>
</tr>
<tr>
<td>A&amp;C</td>
<td>0</td>
</tr>
<tr>
<td>(A&lt;B) &amp; (A&gt;=C)</td>
<td>1</td>
</tr>
</tbody>
</table>

A limited amount of address arithmetic is permitted by means of unary operators \( @ \) and \( % \). Each identifier in a program has associated with it an address and a value stored at
that address. In most high level languages the programmer has access only to the values stored at addresses and not the addresses themselves. One reason for this is that programs typically are not run at fixed locations so that the address associated with any given identifier varies from run to run. An assumption is made with programs written in INTERCOM that load point addresses will be constant with respect to a given compilation.

Suppose for example that LOOP1, LOOP2, and DONE are labels on statements in a program, then the statement

\[
\text{LET } (A(0) = \text{LOOP1}) \& (A(1) = \text{LOOP2}) \& (A(2) = \text{DONE});
\]

establishes an array of addresses. This array might be used in a transfer statement such as

\[
\text{GO TO } %A(N);
\]

where \( N \) represents one of the values 0, 1, or 2. Transfer of execution is not made to one of the locations \( A \), but rather to the location whose address is stored in the \( A \) array. Thus the unary operator \( % \) gives a level of indirect addressing.

Shifting of bits within fields is possible with unary operators \( << \) (left shift) and \( >> \) (right shift). A size factor can be specified for multiple bit shifts. The shifts are zero fill as shown in the examples

<table>
<thead>
<tr>
<th>expression</th>
<th>byte value</th>
</tr>
</thead>
<tbody>
<tr>
<td>( A=5 )</td>
<td>00000101</td>
</tr>
<tr>
<td>( &lt;&lt; A.2 )</td>
<td>00010100</td>
</tr>
<tr>
<td>( &gt;&gt; A.1 )</td>
<td>00000010</td>
</tr>
</tbody>
</table>
Two built-in functions CHAR and HEX are provided for initializing data areas. Values generated by the CHAR function depend on the code in use at a specific installation while the HEX function is independent of such code.

Expressions are strings of identifiers and integer constants separated by operators. There is no implied operator hierarchy and parentheses must be used to specify any altered order of execution.

Conditional

The decision making instruction in INTERCOM is the

\[
\text{IF } \text{exp} \text{ THEN * ELSE *}
\]

construction common to ALGOL, PL/1, and a number of other languages, where exp represents an expression and * represents a statement or block. However, most of these languages require that the tested expression be Boolean or relational since only these have "truth" value. Since all values in INTERCOM have truth value dependent on the low order bit as described earlier, the conditional has greater flexibility.

For example consider the statement

\[
\text{IF } A=(A+1) \text{ THEN stmt1 ELSE stmt2;}
\]

which causes the value represented by A to be incremented and stmt1 is executed if the result is odd, otherwise stmt2 is executed.
Looping

Repeated execution of a block of code is specified by

```
WHILE exp
  BEGIN;
  ---
  END;
```

As long as the expression is "true" as defined earlier the block will be repeatedly executed. The embedded assignment makes this form of looping quite flexible as shown here

```
LET A=1;
WHILE A=(A+2)<13
  BEGIN;
  ---
  END;
```

Control

A HALT command is provided primarily as an aid to debugging a system. This assumes that programs will be run in a stand alone environment with no operating system other than that being tested. The WAIT command causes the system to cease processing until any external interrupt occurs. In the CN INTERRUPT block a RETURN statement must be executed to return control to the next statement beyond the WAIT.

This completes the informal description of the INTERCOM language and structure of programs. The most distinguishing features of the language are summarized in Table 8.
Table 8. Distinguishing features on INTERCOM.

1. Single data type
2. Hierarchical storage
3. Handling of asynchronous input
4. Size factor specification on identifiers
5. Logical arithmetic
6. Global declarations
7. Address manipulation
8. Simple subroutine linkage
9. Embedded assignment operator
10. No operator hierarchy
11. Value oriented relational and Boolean expressions
12. Shift operators
CHAPTER V.
FORMAL DESCRIPTION OF THE LANGUAGE SYNTAX

A computer program consists merely of strings or sequences of symbols. The set of rules which determine which strings are legal combinations is called the syntax of the programming language. The rules which associate meaning with legal strings are the semantics of the language. Syntactic rules for programming languages can often be expressed as a phrase-structure grammar (10, 18, 33). A phrase-structure grammar, G, is a 4-tuple (S, Vn, Vt, P) where S is a starting symbol or head of the language, Vn is a set of non-terminal symbols, Vt is a set of terminal symbols, and P is a set of production rules or rewriting rules. The set of all strings generated by a grammar is called the language. The vocabulary of the language is the union of the set of non-terminal symbols with the set of terminal symbols.

Various schemes have been proposed for describing grammatical rules of programming languages. Among the first and most widely used is the Backus-Naur Form (BNF) used to describe the syntax of ALGOL 60 (3). In order to describe one language in terms of another language certain symbols need to have meaning only in the defining language. These are called metalinguistic symbols. For example in BNF the symbol '::=' means 'is defined to be'. Names which have meaning only in the defining language are sometimes called metalinguistic
variables and are elements of the set $V_n$ mentioned earlier. The ASSIGNMENT statement in INTERCOM can be written in BNF as

$$<\text{ASSIGNMENT}> ::= \text{LET} <\text{EXPRESSION}>$$

In this example $<\text{ASSIGNMENT}>$ and $<\text{EXPRESSION}>$ are metalinguistic variables (elements of $V_n$), LET is a terminal symbol (element of $V_t$), the symbol '$::=' is a metalinguistic symbol (also an element of $V_n$), and the entire string is a production rule (element of $P$).

Parsing a sentence written in a programming language consists of finding the sequence of production rules which will generate the sentence. This sequence of productions need not be unique. A top-down canonical parse starts with the sentence and attempts to find a production rule for the leftmost portion of the sentence. When found the process is repeated on the remaining portion of the sentence string and this cycle continues until the list of productions to generate the entire sentence is derived or the parse fails. A metalanguage similar to BNF called META PI which implements a top-down parsing algorithm has been described by O'Neil (24). META PI has been chosen as the descriptive metalanguage for the syntax of INTERCOM.

The META PI structure is very similar to BNF as shown by the example below of the assignment statement written in META PI.

$$\text{ASSIGNMENT} ::= \text{LET}' \text{EXPRESSION} ;$$
Each META PI statement consists of a metalinguistic variable (non-terminal symbol) followed by ':=', followed by strings of terminal and/or non-terminal symbols and terminated by a semi-colon. In the META PI description of INTERCOM syntax which follows other metalinguistic symbols are described as they are encountered.

**System/Program**

An INTERCOM system is defined by

```
SYSTEM:= 'START' ':'
   CONDITIONS
   PROGRAM
   'FINISH' ':' ;

PROGRAM:= DECLARATIONS STMTLIST;
```

Note that terminal symbols are enclosed in single quotes.

The metasymbol SYSTEM is the starting symbol. In META PI the starting symbol, S, must be defined by the first production rule. A program consists of declarations and a list of statements.

**Expressions**

```
EXPRESSION:= TERM $ (OPERATOR TERM) ;
TERM:=
   (UNARY FACTOR) | FACTOR ;
FACTOR:=
   FUNCTION | IDENTIFIER | .INT
   | ('$' '(' EXPRESSION ')') ;
```

The symbol '$' means 'followed by any number of replications, including zero, of'. Thus, an expression is a term followed
by any number of operator-term pairs. The symbol '|' means 'or' and is used to separate alternatives. A TERM can be a FACTOR preceded by a UNARY operator or just a FACTOR. Parentheses are used for grouping in the familiar sense. The symbol .INT for Integer is satisfied only by a string of decimal digits of arbitrary length. Note that a FACTOR can be an EXPRESSION enclosed in parentheses.

Operators

OPERATOR:= RELATIONAL | NUMERIC ;
RELATIONAL:= '<=' | '>= ' | '== ' | '!= ' | '< ' | '>' ;
NUMERIC:= '=' | '+' | '-' | '*' | '/' |
| '& ' | '|' | 'AND' | 'OR' | '#' ;
UNARY:= '- ' | 'NOT' | '<=' | '>' | '< ' | '>' | '$ ' | '0 ' | '%' ;

Meanings of the operator symbols were given in Chapter IV.

Functions

FUNCTION:= ('HEX' HEXARG) | ('CHAR' CHARARG) ;
HEXARG:= ('HEXPAIRS') ;
HEXPAIRS:= (HEXDIGIT HEXDIGIT) $ (HEXDIGIT HEXDIGIT) ;
HEXDIGIT:= '0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9'
|'A'|'B'|'C'|'D'|'E'|'F' ;
CHARARG:= ('" .TC(') ')') ;

The directive .TO in META PI has as its argument a character in quotes. The directive causes the system to scan the input string until the quoted character is found. In the case of CHARARG above a right parenthesis is sought. Thus, in a character argument all symbols are permitted except a right parenthesis.
Identifiers

IDENTIFIER := NAME (('.' EXPRESSION) | .EMPTY) ;
NAME := .ID QUALIFIER ;
QUALIFIER := SUBSCRIPT | .EMPTY ;
SUBSCRIPT := '(', EXPRESSION ')';

The symbol .EMPTY always returns a true value. Thus, a NAME can be followed by an EXPRESSION, but it need not be. The symbol .ID is satisfied by any string of alphanumeric characters where the first is alphabetic, i.e. .ID is a variable name in the FORTRAN sense.

Declarations

DECLARATIONS := (COMMENT | DECLARES) $ (COMMENT | DECLARES);
COMMENT := "" .TO("") "" ;
DECLARES := 'DECLARE' DECLARELIST ';' ;
DECLARELIST := ENTRY $ (',', ENTRY) ;
ENTRY := NAME (ATRVECTOR | .EMPTY) ;
ATRVECTOR := ATTRIBUTE $ ATTRIBUTE ;
ATTRIBUTE := (('BYTE' | 'HIERARCHY') SUBSCRIPT)
| ('BUFFER' DELIMITER) ;
DELIMITER := 'UNIT' SUBSCRIPT ;

The set of declarations described by the above is an infinite set. Examples of declarations have been given in the previous chapter.

Conditions

CONDITIONS := ':.' .ID '::' 'ON' 'INTERRUPT' QUNITQ
| ('RETURN' '::' | BLOCK) ;
QUNITQ := SUBSCRIPT 'UNIT' SUBSCRIPT ;
Statements

The following defines all of the allowable statement types:

```
STMTLIST := STATEMENT $ (COMMENT | STATEMENT) ;
STATEMENT := ':' .ID ':' UNLABELLED | UNLABELLED ;
UNLABELLED := BLOCK | IFSTATEMENT | LOOP | NONBLOCK ';' ;
BLOCK := 'BEGIN' ';' STMTLIST 'END' ';' ;
IFSTATEMENT := 'IF' EXPRESSION 'THEN' STATEMENT
              (('ELSE' STATEMENT) | .EMPTY) ;
LOOP := 'WHILE' EXPRESSION BLOCK ;
NONBLOCK := ASSIGNMENT | TRANSFER | INOUT
          | CONTROL | ACTIVATE | STORAGE ;
ASSIGNMENT := 'LET' EXPRESSION ;
TRANSFER := JUMP | SUBJUMP ;
JUMP := 'GO' 'TO' TARGET ;
SUBJUMP := 'CALL' TARGET ;
TARGET := ('%' | .EMPTY) NAME ;
INOUT := ('INPUT' | 'OUTPUT') DELIMITER IDENTIFIER ;
CONTROL := 'WAIT' | 'HALT' | 'RETURN' ;
ACTIVATE := ('ENABLE' | 'DISABLE' | 'SIGNAL') .ID ;
STORAGE := ('ALLOCATE' | 'FREE') DECLARELIST ;
```

Based on parsing traces of sample programs using the META PI parsing algorithm, numerous changes in the INTERCOM syntax have been made from its original form. The META PI system was a significant aid in eliminating syntactic errors and ambiguities. The complete syntax of INTERCOM written in META PI is given alphabetically in Appendix A.
Sentences in a given programming language must ultimately have meaning for a computer. For higher level languages the meaning is typically defined by a compiler or interpreter. Rules which govern the meaning of a sentence are the semantics of the language. In the past the syntax of a given language has normally been machine independent while the semantics have been highly machine dependent. Pollack (25) has written, "Formal semantics is one of the least understood subjects in the area of compiler writing and relatively little has been published on the subject. Most semantics systems wind up being somewhat ad hoc and are usually based on a specific interpreter or compiler."

Some progress has been made in the quest for a system in which semantic rules can be defined in a machine independent fashion. Wirth and Weber (33) describe the semantics of the language, EULER, by means of an abstract interpreter system. More recently several authors such as Neuhold (23) and Wegner (30) have written about a system developed at IBM Vienna Labs called the Vienna Definition Language (VDL). The goal of VDL is to provide a means for describing both syntax and semantics of programming languages independent of machine architecture. The underlying principle of VDL, according to Neuhold, "is based on the notion that interpretive execution
of a program in fact constitutes a semantical description of that program."

In this chapter a system based on VDL concepts is developed to describe both syntax and semantics of INTERCOM. Only those VDL features essential for this development are discussed. More detailed information on VDL can be found in the references given.

Computer programs typically contain a certain amount of redundant information. For example in INTERCOM the syntax requires that the keyword, ON, be followed by the keyword, INTERRUPT. The word, INTERRUPT, adds nothing to the computer meaning, but may be necessary as a human factors element. The syntax of INTERCOM stripped of symbols unnecessary for semantic interpretation will be called the abstract syntax of the language. Comments, for example, are absent in such syntax since they add nothing to semantical meaning. The decision-making statement in INTERCOM becomes

\[
\text{IFSTATEMENT := 'IF' EXPRESSION STATEMENT ;}
\]

with the redundant 'THEN' removed. An abstract program is one which satisfies the abstract syntax. An abstract program will be represented by a tree structure. Branches of the tree will correspond to segments of the abstract program. The outermost nodes (called terminal nodes or leaves) of a tree will consist of terminal symbols (also called elementary
components). The tree structure for the statement

```
IF A<B THEN GO TO LOOP ;
```

is given in Figure 4.

![Figure 4. A sample tree structure.](image)

Selectors for components of a given abstract statement will be named and associated with branches of a tree. Selector functions operate on a tree or subtree and return the selected portion. Examples of selector names are shown in Figure 5.

![Figure 5. An abstract tree with selector functions.](image)

The abbreviations used in Figure 5 are:

- **S-ST-ID**: "Select statement identifier"
- **S-EXP**: "Select expression"
- **S-STMT**: "Select statement"
- **S-OPRND1**: "Select operand one"
- **S-RATOR**: "Select operator"
- **S-OPRND2**: "Select operand two"
- **S-TARGET**: "Select target"
If the tree in Figure 5 is given a name, for example x, the use of selector functions can be illustrated by S-ST-ID(x) = 'IF', that is, the statement identifier selected is 'IF'. Note, also, that S-EXP(x) returns a subtree as a value and, hence, S-OGRND2(S-EXP(x)) = 'B'. If the selected object is not found, the null object is returned. It is clear that different selectors must be used for trees that represent different statement types. Trees may be divided into classes according to the selector functions which must be applied. Associated with each class will be a test function called a predicate which will test a tree for class membership. As an example the predicate IS-IF-STMT (read "Is it an IF statement?") when applied to the tree structure, x, in Figure 5 will return a value true, while IS-LET-STMT(S-STMT(x)) will return a false value since S-STMT(x) is a transfer statement (GO TO) not an assignment statement (LET).

Some predicates are true only for elementary objects (keywords or symbols) such as IS-DECLARE which is true only for the keyword 'DECLARE'. The predicates for elementary objects in INTERCOM are listed in Appendix C. Four special predicates for elementary objects will also be used:

- IS-ID True for any valid identifier
- IS-INT True for any integer
- IS-HEX-ARG True for strings of hexadecimal digit pairs
- IS-CHAR-ARG True for any character string terminated by a right parenthesis
Predicates for other than elementary objects can be defined in terms of selectors and other predicates. For example the IF statement predicate can be defined by:

\[
\]

A predicate may be a composite object (built from other predicates) such as

\[
\text{IS-CTL} = \text{IS-WAIT V IS-HALT V IS-RETURN}
\]

which defines control statements. The symbol 'V' is read 'or'. A convention used by Neuhold (23) and also adopted here is to use '-LIST' as suffix for sequences of predicates. For example IS-STMT-LIST is satisfied by a list of elements each of which satisfies the predicate IS-STMT. A test for a PROGRAM in INTERCOM can now be defined with the notation described.

\[
\text{IS-PROGRAM} = (\text{<S-DCLS-LIST: IS-DCLS-LIST>}, \text{<S-STMT-LIST: IS-STMT-LIST>})
\]

The abstract syntax of INTERCOM written in VDL notation is given in Appendices B and C. Abstract programs using this syntax will serve as input to an abstract interpreter developed in the following sections. This development has many parallels in the work of Neuhold (23) and Wegner (30).
An interpreter differs from a compiler in that the former assumes that data, as well as program, is present in storage. Program execution by the abstract interpreter for INTERCOM will proceed as a sequence of state transitions. Each state transition represents an information structure transformation which may involve instruction modification as well as data storage modification. Execution begins with a state designated as the initial state. Instructions will be of two types: macro instructions and value-returning instructions and each instruction will be associated with a terminal node of a tree. A macro instruction replaces its associated node with a subtree and does not modify other components of the state. A value-returning instruction passes a value to its predecessor node and may have side effects of modifying other state components.

An arbitrary state, \( S \), of the interpreter for INTERCOM has the following components:

1. NAMES, with selector \( S\text{-NAMES} \) and predicate \( IS\text{-NAMES} \): a set of generators of unique cell names (addresses) for programs, identifiers, and labels

2. CONTROL TABLE, with selector \( S\text{-CTLTBL} \) and predicate \( IS\text{-CTLTBL} \): a table of program names and their associated cell names, \( P(i) \)

3. ENVIRONMENT, with selector \( S\text{-ENVTBL} \) and predicate \( IS\text{-ENVTBL} \): a table of identifiers and their associated cell names, \( N(j) \)

4. DENOTATION, with selector \( S\text{-DENTBL} \) and predicate \( IS\text{-DENTBL} \): a table of cell names and values linked to identifiers
5. ATTRIBUTE, with selector S-ATTRBL and predicate IS-ATTRBL: a table of cell names and sets of attributes associated with the cells

6. LABELS, with selector S-LBLTBL and predicate IS-LBLTBL: a table of label identifiers and their associated cell names, M(k)

7. LABEL DENOTATION, with selector S-DNLTBL and predicate IS-DNLTBL: a table of cell names and sets of label attributes associated with the cells

8. INTERRUPT, with selector S-INRBTBL and predicate IS-INRBTBL: a table of cell names (for programs) and interrupt attributes for associating interrupt conditions with programs

9. CONTROL NUMBER, with selector S-CTLNUM and predicate IS-CTLNUM: a cell name (number) which at any given time represents the program currently in execution

10. CONTROL, with selector S-CONTROL and predicate IS-CONTROL: a set of cell names and associated programs and control elements for program execution

The state of the INTERCOM INTERPRETER MUST SATISFY:

\[
\text{IS-STATE} = \langle S\text{-NAMES: IS-NAMES}, S\text{-CTLTBL: IS-CTLTBL}, S\text{-ENVTBL: IS-ENVTBL}, S\text{-DENTBL: IS-DENTBL}, S\text{-ATRTBL: IS-ATRTBL}, S\text{-LBLTBL: IS-LBLTBL}, S\text{-DNLTBL: IS-DNLTBL}, S\text{-INRBTBL: IS-INRBTBL}, S\text{-CTLNUM: IS-INT}, S\text{-CONTROL: IS-CONTROL} \rangle
\]

A DUMP component found in other VDL interpreters in the literature is not needed here since variables are global in scope and block entry/exit does not require saving environment information. Part of the functions of a DUMP component have been retained as a STACK component of the CONTROL tree for linking to a subprogram as is explained later.
The tree structure representing the STATE is shown in Figure 6.

**Figure 6. Components of the STATE of the interpreter.**

Abstract execution of an INTERCOM program by the interpreter will consist of transformations on the information structure represented by STATE. Hence, the state components require more formal definition which will now be given in VDL notation.

Program names, identifiers, and labels will be associated with cell names (addresses) by the NAMES component. At any given time these components specify indices i, j, and k of the next unique names to be generated. The cell names are merely integers. Formally the syntax is

\[
\text{IS-NAMES} = (\langle S-\text{PROG-NUM}: \text{IS-INT} \rangle, \\
\langle S-\text{ID-NUM}: \text{IS-INT} \rangle, \\
\langle S-\text{LBL-NUM}: \text{IS-INT} \rangle)
\]
Each INTERCOM program can be considered as a collection of "programs" with the body of the program as one segment and each ON condition as other segments. The body segment will be the MAIN program and each ON condition block will be a subordinate program and referenced by its associated label. In the remainder of this paper the ON condition blocks will be considered as semi-independent programs.

The CONTROL TABLE associates program names with locations and is formally defined by

$$\text{IS-CTLTLB} = \{\langle \text{PGM: IS-PR0G-NUM} \rangle | \text{IS-ID(PGM)}\}$$

This notation is read as "the set of all name-number pairs where the selector PGM (name portion) satisfies the predicate IS-ID and the second element (number portion) satisfies the predicate IS-PR0G-NUM." The table enables referencing of programs by location. Its basic structure is shown in Figure 7 below.

![Diagram of IS-CTLTLB](attachment:image)

Figure 7. Structure of the CONTROL table.
The ENVIRONMENT component is very similar to the CONTROL TABLE except that identifiers rather than program names are associated with cells.

\[
\text{IS-ENVTLB}=\{<\text{ID}: \text{IS-ID-NUM}> \mid \text{IS-ID}\}\}
\]

The LABELS component also has the same form

\[
\text{IS-LELTBL}=\{<\text{LBL}: \text{IS-LBL-NUM}> \mid \text{IS-ID(LBL)}>\}
\]

The DENOTATION table differs slightly in that cell names are associated with values.

\[
\text{IS-DENTBL}=\{<\text{N}: \text{IS-INT}> \mid \text{IS-ID-NUM(N)}>\}
\]

In the table for LABEL DENOTATION a pair of values is associated with each cell.

\[
\text{IS-DNLTBL}=\{<\text{L}: \text{IS-LOC}> \mid \text{IS-LBL-NUM(L)}>\}
\]

\[
\text{IS-LOC}=\langle\text{S-STNUM}: \text{IS-INT}>,
\text{S-PFGNUM}: \text{IS-INT}>\}
\]

Each label is associated with a statement number (STNUM) and a program number (PRGNUM) to aid in implementation of cross-program branching.

Cell names may be associated with multiple attributes as indicated by the ATTRIBUTE component.

\[
\text{IS-AITBL}=\{<\text{N}: \text{IS-TYPE}> \mid \text{IS-ID-NUM(N)}>\}
\]

\[
\text{IS-TYPE}=\langle\text{S-SIZE}: \text{IS-INT}>,
\text{S-HIER}: \text{IS-INT}>,
\text{S-UNIT}: \text{IS-INT}>\}
\]

The SIZE entry specifies the number of contiguous bytes to be associated with a given cell name. HIER indicates the level of storage and UNIT specifies the device number to be associated with this cell for unsolicited input. The UNIT
component may be null. Tree structure for the ATTRIBUTE component is shown in Figure 8.

![Diagram]

Figure 8. Structure of the ATTRIBUTE table.

The CONTROL NUMBER and INTERRUPT components are defined by

\[
\text{IS-CTLNUM} = \langle \text{S-CTLNUM: IS-INT} \rangle
\]

\[
\text{IS-INTRTBL} = (\{ \langle P: \text{IS-INRATS} \rangle \text{ IS-PROG-NUM}(P) \})
\]

\[
\text{IS-INTRATR} = (\langle \text{S-DEVICE: IS-INT} \rangle,
\text{<S-LEVEL: IS-INT>},
\text{<S-PVAL: IS-PROG-NUM}>)
\]

The INTERRUPT table serves as a link to an external environment. For each program it contains three entries: the pro-
gram number, the device number, and the interrupt number.

The CONTROL NUMBER is initially 1 representing the MAIN program in execution. CTNUM is changed by execution of a CALL, GC TO, or SIGNAL statement as well as any external interrupt. It is assumed that an external interrupt system uses INTRTBL to determine a value for CTLNUM. Allowing external conditions to effect CTLNUM and hence program execution enhances the interpreter as a model for asynchronous activity.

The CONTROL component is by far the most complex, primarily because it contains the statements to be executed. Formally it is defined as

\[
\text{IS-CONTROL} = (\{ \text{P: IS-SEGMENT} \mid \text{IS-PROG-NUM(P)} \})
\]

\[
\]

\[
\text{IS-STKLELEM} = (\{ \text{S-RTN-STC: IS-INT}, \text{S-RTN-PROG: IS-INT} \})
\]

The predicates IS-DCLS-LIST and IS-STMT-LIST are satisfied respectively by lists of declarations and statements of programs written in INTERCOM and are defined in Appendix B. The declarations component of ON condition blocks will be null. Each program has its own statement counter (STC) and its own stack (STACK). When a program calls a labelled block (e.g. CALL TEST) the stack of the "called" program receives the statement counter (RTN-STC) and the unique program number (RTN-PBCG) of the "calling" program. Execution of a RETURN
statement within the block will retrieve this information from the stack and execution will return to the calling program. If no RETURN statement is executed, the CALL has the same effect as the GO TO except that the stack retains unused entries. The stack contains number pairs and the top of the stack can be referenced by S-STKELEM.

This completes the formal description of the structure of the abstract interpreter. Actions taken by the interpreter on INTERCOM programs will be described in the following sections.

Interpreter rules have the following format

\[
\text{instruction-name}(\text{parameters}) = \\
\quad \text{cond}1 \text{ then instruction-list}_1 \\
\quad \text{cond}2 \text{ then instruction-list}_2 \\
\quad \text{...} \\
\quad \text{cond}N \text{ then instruction-list}_N
\]

Conditional expressions represented by condN are tested top to bottom and the first condition which is true causes the corresponding instruction list to be added to the execution control tree. The execution control tree assumes execution from bottom_to_top so that within an instruction list execution is from bottom_to_top. This is quite important when reading the interpreter rules. The format is perhaps best illustrated by example.
II. "interpret system"

\[\text{int-system}(S) = \]

(7) \(\text{PASS} := \text{null};\)
(6) \(\text{int-stmt-list}(t);\)
(5) \(t: \text{S-CONTROL}(\text{P}(\text{S-CTLNUM}(S)))\)
(4) \(\text{S-CTLNUM} := 1;\)
(3) \(\{\text{scan}(p) | 2 \leq p < \text{sys-length}(\text{S-COiîTEOL}(S))\};\)
(2) \(\text{create-tables}(h, 1);\)
(1) \(\text{int-dcls}(\text{S-DCLS-LIST}(k));\)

\[\text{k:S-CONTROL}(\text{P}(\text{S-CTLNUM}(S)))\]

The initial state of the INTERCOM interpreter has int-system(S) as its only node on the execution control tree. All other nodes grow from this one node. The CTLNUM component is assumed to be 1 and also values of the name generators, statement counters, and activation flags are 1. Execution of int-system(S) causes the execution control tree to be replaced by its instruction list. The next instruction (1) \(\text{int-dcls}(\ldots)\) causes its node to be replaced by its instruction list (see I2 in Appendix D). Thus, many instructions may be executed before execution control returns to instruction (2) in int-system(S). Parameters of instructions may be defined by selectors as in

\[\text{k: S-CONTROL}(\text{P}(\text{S-CTLNUM}(S)))\]

The colon means "is defined by". In this case k is a program tree selected by the current value of CTLNUM. Here S-CTLNUM(S) acts as a subscript on P to fully qualify P as a selector for a specific program tree. Since k is a program tree then S-DCLS-LIST(k) selects the declarations list compo-
ment of that tree. The notation

\[ S-CONTROL(P(S-CTLNUM(S))) \]

is used throughout to specify selection of a specific program tree. Information may be passed to the predecessor node by \( PASS:= \) value or tree.

The conditional expressions may be made unconditional by specifying \( TRUE \) as the condition. The statement

\[ \{ \text{scan}(pc) \mid 2 \leq pc \leq \text{sys-length}(S-CONTROL(S)) \} ; \]

is an iteration statement which calls the instruction, scan, for integer values 2 thru the length of the system (sys-length).

The interpreter rules will now be described briefly and new notation explained as it is encountered. The formal description of the interpreter is given in Appendix D and should be used in conjunction with the prose descriptions.

II. "interpret system"

\[ \text{int-system}(S) = \]

Parameter \( S \) represents the initial state of the abstract interpreter. The sequence of interpretation for this instruction is:

(1) interpret all declarations in the MAIN program, which builds the ENVIRONMENT and ATTRIBUTE tables

(2) build the LABEL and LABEL DENOTATION tables for the MAIN program

(3) increment the CTLNUM to move to the first ON-condition
block

(4) build the CONTROL TABLE and INTERRUPT table for the ON-
conditions; also, build the LABEL and LABEL DENOTATION tables
(5) reset CTLNUM
(6) execution of statements in the MAIN program now begins
and continues until the statement list is exhausted
(7) null replaces the node on the execution control tree
and execution ceases

12. "interpret declarations list"

\[ \text{Int-dcls-list}(t) = \]
Parameter \( t \) is the tree representing the list of declarations. This instruction is called recursively until all declarations have been executed. The statement counter is incremented so that program execution continues after the declarations have been processed.

13. "head of a tree"

\[ \text{Head}(t) = \]

14. "tail of a tree"

\[ \text{Tail}(t) = \]

Head and tail of a list are discussed by Wegner (30).

15. "interpret declarations"

\[ \text{Int-dcls}(t) = \]
The parameter \( t \) in this case is a single declaration statement tree which must be subdivided into entries.
I6. "interpret entry list"

\texttt{Int-entry-list(t)=}
The \( t \) is a list of entries and this instruction is also executed recursively until each entry has been processed.

I7. "interpret entry"

\texttt{Int-entry(t)=}
Parameter \( t \) is a single entry which includes an identifier and its attributes. This instruction invokes a table building instruction for each element of an array in the case of a subscripted identifier.

I8. "update environment and attribute tables"

\texttt{Uptbls(t)=}
Parameter \( t \) is a single entry in a declarations statement.

I9. "create tables"

\texttt{create tables(a,z)=}
Parameter \( a \) is a program tree and \( z \) is an initial value for the iteration parameter. A build instruction is executed for each statement in the tree \( a \).

I10. "build label and denotation label tables"

\texttt{build(a,b)=}
Parameter \( a \) is a program tree and \( b \) is an iteration value that selects a single statement, \texttt{elem(b)}, from the program
tree. If the statement is labelled, then both LBLTBL and
DNLTBL are updated.

111. "increment control number"

up-ctlnum=
This parameterless instruction increments the CTLNUM compo-
nent which effectively changes execution from one program to
another.

112. "increment statement counter"

up-stc=
The statement counter component of the program designated by
CTLNUM is incremented.

113. "build control and interrupt tables"

scan(i)=
Parameter i is a program number so that P(i) is a program
tree. This instruction builds CTLTBL and INTRTBL and invokes
building of LBLTBL and DNLTBL for ON-condition blocks.

114. "generate program number"

gen-prog-num=
A unique program number (address) is generated and the NAMES
component of the STATE is modified.

115. "generate identifier number"

gen-id-num=
A unique address to be associated with an identifier is gen-
erated and the NAMES component is modified.

I16. "generate label number"

gen-lbl-num=

A unique cell number associated with a label is generated and the NAMES component is modified. Instructions I14 and I15 do not return values but merely modify a component of the STATE.

I17. "update control table"

update-ctltbl(c)=

Parameter c represents a statement tree. This instruction introduces a powerful tree construction operator, mu. In this case mu operates on CTLTEL and searches for a given label. If found the branch is replaced by a new selector-object pair <pgm;p>.

I18. "update interrupt table"

update-intrtbl(c)=

Parameter c is a statement tree. A special case of the mu operator, mu0, is used to create a tree of three selector-object branches. The mu0 operator does no searching, but merely replaces a node with a tree. In this case the interrupt table is updated. The mu operators are discussed by Wegner (30).

I19. "update denotation label table"

update-dnltbl(t)=
Parameter t is a program tree. DNLTBL is updated with a new
tree entry to associate labels with the appropriate program
and statement counter.

I20. "update label table"
update-lbltbl(u) =
The parameter u is a statement tree. A labelled statement
has been encountered and the label table is modified
accordingly.

I21. "update attribute table"
update-atrtbl(r) =
Parameter r is an entry in a declaration. The attribute
table is modified by a new subtree of attributes for the
identifier in r.

I22. "find size"
find-size(b) =
I23. "find hierarchy"
find-hier(b) =
I24. "find unit"
find-unit(t) =
These three instructions have as parameter, b, the attribute
portion of a declaration entry and return the value of the
expression associated with each attribute. If an attribute
is absent, a null value is returned.
I25. "update denotation table"

update-dentbl(id, val) =

Parameter id represents an identifier for which the search is to be made and val is the value to associate with the identifier. Before the DENTBL can be updated the address associated with id must be found by searching the LBLTBL.

I26. "update environment table"

update-envtbl(r) =

Parameter r is a statement tree. The environment table is updated with a new id-address pair.

I27. "determine system length"

sys-length(S) =

The parameter S is the current STATE and the value returned is the number of SEGMENTS (MAIN program + #ON-conditions) in the system.

I28. "determine program length"

prog-length(t) =

Parameter t is an arbitrary tree and this instruction returns the number of immediate branches.

I29. "interpret statement list"

int-stmt-list(t) =

Parameter t is the statement list associated with a specific
program. This instruction is the heart of program execution and coordination. As long as the CTLNUM and statement counter (STC) remain in range this instruction is recursively executed. If the counters mentioned are out of range the null tree is returned.

I30. "interpret statement"

int-stmt(t) =

Parameter t is a statement tree. This instruction dispatches statement execution by statement type.

I31. "interpret block"

int-block(t) =

Parameter t is a statement list tree (i.e. block).

I32. "interpret IF-THEN statement"

int-if(t) =

The parameter t is an IF-THEN statement. An expression is tested and if true the statement in the THEN clause is executed, otherwise it is bypassed and the statement counter is incremented accordingly.

I33. "interpret IF-THEN-ELSE statement"

int-else(t) =

This instruction is similar to I32 except that t represents an IF-THEN-ELSE statement. In this case the ELSE clause must be bypassed if the THEN clause is the alternative chosen.
after expression testing.

I34. "true/false test"

test(e) =

Parameter e is the value of an expression. If it is odd, TRUE is returned otherwise FALSE is returned.

I35. "interpret assignment statement"

int-assign(t) =

Parameter t is an assignment statement. Only expression evaluation is necessary.

I36. "interpret transfer statement"

int-transfer(t) =

Parameter t is either a GO TO or CALL statement. Execution requires that the target label be found.

I37. "find target in label table"

find-target(t, id) =

Parameter t is the LBLTAB. This instruction is called recursively until the identifier, id, is found or the table is exhausted. It returns the address associated with id.

I38. "execute a jump"

exec-jump(tgt, t) =

Parameter tgt is the cell number associated with an id. If this instruction is invoked by execution of a GO TO then parameter t is null and stacking of statement counter and pro-
gram number is bypassed.

I39. "stack current statement counter and program number"

stknum(tgt) =

Parameter tgt is a cell number. This instruction stacks the statement counter and program number of the calling program in the stack of the called program.

I40. "move control to target statement"

move-ctl(tgt) =

Parameter tgt is the cell number associated with the label to which transfer is being made. The new CTLNUM and statement counter are established and execution continues with the new statement possibly in a different program.

I41. "interpret input/output statement"

int-inout(t) =

Parameter t is an input or output statement. Three values are passed to externally defined routines; the address where input/output is to start, the scale factor in case multiple bytes are used, and the unit number of the input/output device.

I42. "interpret control statement"

int-ctl(t) =

Parameter t is a WAIT, HALT, or RETURN statement. In the case of WAIT this instruction recursively calls itself until
an external interrupt changes the statement counter value. In the case of HALT the CTLNUM is set out of range so that statement execution ceases.

I43. "interpret return statement"

t-return (t) =
Parameter t is the RETURN statement. The statement counter and CTLNUM are retrieved from the stack and execution control is passed to those values.

I44. "interpret activate statement"

int-activate (t) =
The parameter t is an ENABLE, DISABLE, or SIGNAL statement. In the case of ENABLE or DISABLE an instruction is invoked to set the activation flag. For the SIGNAL statement a call is made to simulate an interrupt.

I45. "set activation flag"

set-flag (t, f) =
Parameter t is the statement and f is the flag value for the activation component.

I46. "Simulate an interrupt"

sim-intr (t) =
Parameter t is the SIGNAL statement. First the activation flag is tested and control passes to the target only if the activation flag is set, otherwise execution continues with
the next statement. This instruction acts nearly the same as a CALL statement.

I47. "interpret storage allocation"

\[ \text{int-dstore}(t) = \]

Parameter \( t \) is an ALLOCATE or FREE statement. If \( t \) is an ALLOCATE statement it is treated the same as a declaration statement. The data structure defined is not well designed to handle release of storage dynamically. In this case the identifier is merely removed from the ENVIRONMENT table and its associated value is deleted from DENTBL.

I48. "interpret free storage entry list"

\[ \text{int-stgentry}(t) = \]

Parameter \( t \) is a list of entries in the FREE statement. This instruction is called recursively until all entries are processed. It invokes the int-free instruction.

I49. "interpret FREE statement"

\[ \text{int-free}(t) = \]

Parameter \( t \) is an entry in the list. If the identifier is subscripted this routine iterates until all table entries are deleted for each subscript.

I50. "delete table entries"

\[ \text{deltbls}(t) = \]

Parameter \( t \) is an entry in the list. The ENVIRONMENT and
DENOTATION tables are updated by deleting entries.

I51. "interpret looping statement"
int-loop(t) =
Parameter t is a WHILE statement. As long as the value of the expression is true the associated block is executed and int-loop is called recursively. When the expression is FALSE the block is bypassed.

I52. "skip past while block"
skip-block(t) =
Parameter t is a WHILE statement. This routine causes execution of the associated block to be bypassed.

I53. "interpret conditions statement"
int-conditions(t) =
Parameter t is an ON-conditions statement. Either a RETURN is executed or a BLOCK depending on statement type.

I54. "find identifier in environment table"
find-id(id) =
Parameter id is an identifier name. This instruction searches for the identifier in the ENVIRONMENT table and returns its associated address.

I55. "expression evaluation"
eval(t) =
Parameter t is the expression to be evaluated and may contain
subexpressions. The appropriate instruction is invoked depending on expression type.

156. "binary operation evaluation"

\[ \text{int-bin}(\text{op}, \text{oprnd1}, \text{oprnd2}) = \]

Parameter \text{op} is the operation symbol, \text{oprnd1} and \text{oprnd2} are the addresses of the operands. This instruction invokes actual execution.

157. "unary operation evaluation"

\[ \text{int-unary}(\text{op}, \text{oprnd1}) = \]

Details are the same as for 153 except only a single operand is used.

158. "interpret factor"

\[ \text{int-factor}(t) = \]

Parameter \text{t} is the factor under consideration. In the case \text{t} is an identifier its address is returned. If \text{t} is a function or constant it is added to the \text{ENVIRONMENT} and \text{DENOTATION} tables and an address is returned. If \text{t} is itself an expression the eval instruction is called recursively to evaluate \text{t}.

159. "interpret function"

\[ \text{int-function}(t) = \]

Parameter \text{t} is the function. This is the instruction which invokes the actual building of tables mentioned in 155.
I60. "add constants to tables"

```
add-con(t) =
```

Parameter t is the integer, hex, or character constant. This instruction modifies DENTBL and invokes a routine to modify ENVIBL.

I61. "get a location for the constant"

```
get-loc=
```

This instruction returns a unique location name (address) for storing constant values.

I62. "apply the operator"

```
apply(a, op, b) =
```

Parameters a and b are addresses of operands and op is the operation to be performed. This instruction is largely machine dependent and thus will not be further defined here. Wegner (30) discusses expression evaluation and an "apply" instruction in some detail.

This completes the prose description of the formal definition of the abstract interpreter given in Appendix D. In summary the basic concepts of VDL notation have been given, a data structure for an abstract interpreter was described, and finally, instruction rules for the interpreter were defined and discussed.
CHAPTER VII.
A SAMPLE DATA COMMUNICATIONS ENVIRONMENT

To illustrate how the INTERCOM language might be used, a simple data communications environment is specified in this chapter and an example of a software system to handle message traffic in that environment is given in Appendices E and F.

Assume that a small processor is to perform line control functions for two typewriter terminals and that information from these terminals is to be forwarded to a central computing system for processing. This configuration is shown in Figure 9.

Numerous assumptions are made about the characteristics of this system so that complete algorithms can be written. The terminals are assumed to be unbuffered and transmission to the CP is asynchronous and full-duplex (receive and transmit can occur simultaneously). If either terminal transmits more than 72 characters to the CP without sending a message termination character, a buffer overflow condition exists and the CP sends an appropriate message to the termi-
nal. A message is buffered by the CP and translated to EBCDIC code before being passed to the central computer.

Four types of interrupts from the terminals are recognized by the CP as well as one type of interrupt from the central computer. A summary of unit associations and interrupt types is given in Table 9. Each character input from or output to a terminal causes a CP interrupt while only each message from the central computer causes an interrupt.

Table 9. Unit associations and interrupt types.

<table>
<thead>
<tr>
<th>Terminal #1 unit(1)</th>
<th>interrupt(1)</th>
<th>character input interrupt</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>interrupt(2)</td>
<td>attention interrupt</td>
</tr>
<tr>
<td></td>
<td>interrupt(3)</td>
<td>disconnect interrupt</td>
</tr>
<tr>
<td></td>
<td>interrupt(4)</td>
<td>character output interrupt</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Terminal #2 unit(2)</th>
<th>interrupt(1)</th>
<th>character input interrupt</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>interrupt(2)</td>
<td>attention interrupt</td>
</tr>
<tr>
<td></td>
<td>interrupt(3)</td>
<td>disconnect interrupt</td>
</tr>
<tr>
<td></td>
<td>interrupt(4)</td>
<td>character output interrupt</td>
</tr>
</tbody>
</table>

| Central computer unit(3) | interrupt(6) | message output interrupt |

Each message from the central computer has a destination specified as a unit number in the first byte of the text portion of the message. An End of Transmission (EOT) character marks the end of the message and is also transmitted to the terminal.

The CP will perform a limited amount of editing for the terminals in the form of recognizing a special character.
("backspace") to delete the previous character input and a message delete character to delete an entire message. In the event of buffer overflow the current message is deleted as if a message delete character had been transmitted. An attention interrupt while the terminal is in the process of input will act as a line delete. However, attention during output will terminate the output of that message by the CP.

Since a message from the central computer must be transmitted to a terminal on a character by character basis, the central computer must be notified when this transmission is complete. Similar notification must be made in cases of attention interrupt or disconnect. For this reason two types of messages will be sent to the central computer, namely text and control. Text messages will begin with a Start of Text (STX) character and control messages will begin with a Start of Header (SOH) character.

Both text and control messages have as the second byte of a message a source unit specification and messages terminate with an EOT character. The third byte of a control message is:

1 = output done  
2 = attention interrupt  
3 = disconnect
The form for a control message then is

SOH unit ctl EOT

where ctl is given by 1, 2, or 3 above and the form for a text message is

STX unit text EOT

A general flow of control of such a system is given in Appendix E. Some additional simplifying assumptions must be made. First, there is no error detection in the case of transmission errors and second, there is no clock for timing. A clock could be added to the system conceptually by defining a new interrupt type (assuming the clock causes an interrupt). Third, input/output to the central computer is handled by an independent subsystem and the time required to send a complete message to the central computer is less time than to transmit a single character from the terminal. This in effect avoids problems of data overruns which must be considered in a production system, but introduces a level of complexity beyond the scope of this description.

The purpose of defining a simple data communications system and flow of control is to provide a framework for a program written in INTERCOM to illustrate language structure and some language features. The INTERCOM program is given in Appendix F.
CHAPTER VII
SUMMARY AND CONCLUSIONS

From recent literature and industry announcements it seems likely that data communications will have a significant and lasting impact on computer systems. Programmable communications controllers will probably play an even more important role in computing in the near future than they are currently. Software for these controllers will likely be developed in an ad hoc fashion unless high level languages help to standardize the algorithms.

In this paper a high level language for data communications systems has been developed and both syntax and semantics have been described in considerable detail. The structure of the language is somewhat simpler than many, made possible primarily because it is a special purpose rather than a general purpose systems programming language. The absence of local variables in blocks and the absence of parameters in subprogram calls contribute to the simplicity. Facilities for handling asynchronous input/output make the language somewhat unique and enhance its usefulness in a data communications environment. In addition to some new language constructs, features from a variety of different languages have been brought together in a single language. However, language constructs were carefully chosen to avoid violating the basic premise of language simplicity. The choice of language
constructs makes the INTERCOM language unique.

Some potential weaknesses of the language should be noted. The inability to develop a number of independent routines and combine them as a single system may limit its usefulness in constructing large systems. Stack manipulation operations are perhaps a desirable feature absent in the language. The need for a run-time symbol table may create special problems in time-critical algorithms. The BUFFER attribute would be more powerful as a dynamic attribute so that multiple buffers or a pool of buffers could be used for unsolicited input.

Several avenues of future research are suggested by this work. First, a full implementation of the language would aid in the study of the above potential weaknesses. Also, an implementation might help answer questions such as, "Does a top-down left to right parsing algorithm lead to good code generation for this language?" and "Is the language well-suited for syntax directed compilation?". Additional programming experience might answer questions as "Is the language sufficient to describe a large and complex system?" and "Is user specification of hierarchical data storage a desirable language feature?". The VDL notation seems somewhat tedious in describing semantics of a language of this type. Alternative schemes for representing interpreters could be explored.
It is felt that this paper contributes to the field of computer science by exhibiting a new special purpose programming language and by demonstrating that the language can be used in describing data communications algorithms. Further, it shows that complex language structures are unnecessary for restricted classes of systems programming problems.
BIBLIOGRAPHY


ACKNOWLEDGMENTS

The support and encouragement of the faculty of Iowa State University has been greatly appreciated. Special thanks go to Dr. Robert M. Stewart whose seminar helped launch the initial writing; to Dr. Clair G. Maple for the opportunity to develop a software system for a communications processor and who first stimulated my interest in that area; and, to Dr. Roy F. Keller for directing my research and providing valuable advice in its development.
APPENDIX A.

THE SYNTAX OF INTERCOM IN META PI

SYSTEM := 'START' ':' CONDITIONS PROGRAM 'FINISH' ':' ;
PROGRAM := DECLARATIONS STMTLIST ;
ACTIVATE := ('ENABLE' | 'DISABLE' | 'SIGNAL') .ID ;
ASSIGNMENT := 'LET' EXPRESSION ;
ATRVECTOR := ATTRIBUTE $ ATTRIBUTE ;
ATTRIBUTE := ("BYTE" | "HIERARCHY") SUBSCRIPT | ('BUFFER' DELIMITER) ;
BLOCK := 'BEGIN' ':' STMTLIST 'END' ':' ;
CHARARG := "(" .TO("") ")" ;
COMMENT := "" .TO("" ")" ;
CONDITIONS := ':' .IF ':' .ON 'INTERRUPT' QUNITQ ('RETURN' ':' | BLOCK) ;
CONTROL := 'WAIT' | 'HALT' | 'RETURN' ;
DECLARATIONS := (COMMENT | DECLARES) $ (COMMENT | DECLARES) ;
DECLARES := 'DECLARE' DECLARELIST ':' ;
DECLARELIST := ENTRY $ (',', ENTRY) ;
DELIMITER := 'UNIT' SUBSCRIPT ;
ENTRY := NAME (ATRVECTOR | .EMPTY) ;
EXPRESSION := TERM $ (OPERATOR TERM) ;
FACTOR := FUNCTION | IDENTIFIER | .INT |
| ('(' EXPRESSION ')') ;
FUNCTION := ('HEX' HEXARG) | ('CHAR' CHARARG) ;
HEXARG := "(" HEXPAIRS ")" ;
HEXPAAIRS := (HEXDIGIT HEXDIGIT) $ (HEXDIGIT HEXDIGIT) ;
HEXDIGIT := '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
| 'A' | 'E' | 'C' | 'D' | 'E' | 'F' ;
IDENTIFIER := NAME ("." EXPRESSION) | .EMPTY ;
IFSTATEMENT := 'IF' EXPRESSION 'THEN' STATEMENT
(ELSE STATEMENT) | .EMPTY ;
INOUT := ('INPUT' | 'OUTPUT') DELIMITER IDENTIFIER ;
JUMP := 'GO' 'TO' TARGET ;
LCOF := 'WHILE' EXPRESSION BLOCK ;
NAME := .ID QUALIFIER ;
NCELLOCK := ASSIGNMENT | TRANSFER | INOUT
| CONTROL | ACTIVATE | STORAGE ;
NUMERIC := (= | '+' | '-' | '*' | '/'
| '&' | '|' | 'AND' | 'OR' | '#' ;
OPERATOR := RELATIONAL | NUMERIC ;
QUALIFIER := SUBSCRIPT | .EMPTY ;
UNITQ := SUBSCRIPT 'UNIT' SUBSCRIPT ;
RELATIONAL := <= | >= | == | != | < | > ;
STATEMENT := ":" .IL "UNLABELLED" UNLABELLED ;
STMTLIST := STATEMENT $ (COMMENT | STATEMENT) ;
STORAGE := ('ALLOCATE' | 'FREE') DECLARELIST ;
SUBJUMP := 'CALL' TARGET ;
SUBSCRIPT := "(" EXPRESSION ")" ;
TARGET := ("%" | .EMPTY) NAME ;
TERM := (UNARY FACTOR) | FACTOR ;
TRANSFER := JUMP | SUBJUMP ;
UNARY := '!' | 'NOT' | '<<' | '>>' | '$' | '@' | '%' ;
UNLABELLED := BLOCK | IFSTATEMENT | LOOP | NONBLOCK ';;' ;
APPENDIX B.
THE ABSTRACT SYNTAX OF INTERCOM IN VDL

IS-SYSTEM = (<S-ON-CONDITIONS: IS-CONDITIONS-LIST>,
              <S-MAIN-PROGRAM: IS-PROGRAM>)

IS-PROGRAM = (<S-DCLS-LIST: IS-DCLS-LIST>,
                <S-STMT-LIST: IS-STMT-LIST>)

IS-DCLS = (<S-ST-ID: IS-DECLARE>,
            <S-ENTRY-LIST: IS-ENTRY-LIST>)

IS-ENTRY = IS-ATTR-NAME V IS-NAME

IS-NAME = IS-QUALIF-NAME V IS-UNQUALIF-NAME

IS-QUALIF-NAME = (<S-ID: IS-ID>,
              <S-SUBSCRIPT: IS-EXP>)

IS-UNQUALIF-NAME = <S-ID: IS-ID>

IS-ATTR-NAME = (<S-NAME: IS-NAME>
                <S-ATTRIB-LIST: IS-ATTRIB-LIST>)

IS-ATTRIB = (<S-ATTR-TYPE: IS-ATTR-TYPE>,
              <S-EXP: IS-EXP>)

IS-ATTR-TYPE = IS-BYTE V IS-HIERARCHY V IS-BUFFER

IS-EXP = IS-TERM V IS-BIN

IS-TERM = IS-UNARY-FACTOR V IS-FACTOR

IS-BIN = (<S-OPRND1: IS-TERM>,
          <S-RATOR: IS-RATOR>,
          <S-OPRND2: IS-TERM>)

IS-UNARY-FACTOR = (<S-UNARY: IS-UNARY>,
                   <S-FACTOR: IS-FACTOR>)

IS-FACTOR = IS-FUNCTION V IS-IDENT V IS-INT V IS-EXP

IS-RATOR = IS-RELOPR V IS-NOOPR

IS-RELOPR = IS-LSEQ V IS-GBEQ V IS-EQ V IS-NE
            V IS-LSTHAN V IS-GRTTHAN
IS-UNARY = IS-MINUS V IS-NOT V IS-LSHIFT V IS-RSHIFT V IS-COMP V IS-ADR V IS-INDIRECT


IS-FUNCTION = IS-CHAR-TYPE V IS-HEX-TYPE

IS-CHAR-TYPE = (<S-CHAR: IS-CHAR>,
                <S-CHAR-ARG: IS-CHAR-ARG>)

IS-HEX-TYPE = (<S-HEX: IS-HEX>
                <S-HEX-ARG: IS-HEX-ARG>)

IS-CONDITIONS = (<S-LABEL: IS-ID>,
                  <S-ST-ID: IS-ON>,
                  <S-EXP1: IS-EXP>,
                  <S-EXP2: IS-EXP>,
                  <S-BODY: IS-BODY>)

IS-BODY = IS-RETURN V IS-BLOCK

IS-BLOCK = (<S-ST-ID: IS-BEGIN>,
             <S-STMT-LIST: IS-STMT-LIST>,
             <S-END: IS-END>)

IS-STMT = IS-LABEL-STMT V IS-UNLAB-STMT

IS-LABEL-STMT = (<S-LABEL: IS-ID>,
                 <S-UNLAB-STMT: IS-UNLAB-STMT>)

IS-UNLAB-STMT = IS-BLOCK V IS-IF-STMT V IS-LOOP V IS-NONBLK

IS-IF-STMT = IS-IF-ELSE V IS-IF-THEN

IS-IF-THEN = (<S-ST-ID: IS-IF>,
               <S-EXP: IS-EXP>,
               <S-STMT: IS-STMT>)

IS-IF-ELSE = (<S-IF-THEN: IS-IF-THEN>,
               <S-ELSE-CLAUSE: IS-ELSE-CLAUSE>)

IS-ELSE-CLAUSE = (<S-ST-ID: IS-ELSE>,
                   <S-STMT: IS-STMT>)

IS-IFC = (<S-ST-ID: IS-WHILE>,
           <S-EXP: IS-EXP>,
           <S-BLOCK: IS-BLOCK>)
IS-NONBLK = IS-ASSIGN-STMT V IS-TRANSFER V IS-INOUT
V IS-CTL V IS-ACTIVATE-STMT V IS-DSTORE

IS-ASSIGN-STMT = (<S-ST-ID: IS-LET>,
<S-EXP: IS-EXP>)

IS-TRANSFER = (<S-ST-ID: IS-BRANCH>,
<S-TARGET: IS-TARGET>)

IS-BRANCH = IS-GO V IS-CALL

IS-TARGET = IS-NAME V IS-INDIRECT

IS-INDIRECT = (<S-PERCENT: IS-PERCENT>,
<S-NAME: IS-NAME>)

IS-INOUT = IS-INPUT-STMT V IS-OUTPUT-STMT

IS-INPUT-STMT = (<S-ST-ID: IS-INPUT>,
<S-EXP: IS-EXP>,
<S-IDENT: IS-IDENT>)

IS-OUTPUT-STMT = (<S-ST-ID: IS-OUTPUT>,
<S-EXP: IS-EXP>,
<S-IDENT: IS-IDENT>)

IS-IDENT = IS-SCALED V IS-NAME

IS-SCALED = (<S-NAME: IS-NAME>,
<S-SCALE: IS-EXP>)

IS-CTL-STMT = <S-ST-ID: IS-CTL>

IS-CTL = IS-WAIT V IS-HALT V IS-RETURN

IS-ACTIVATE-STMT = (<S-ST-ID: IS-ACTIVATE>,
<S-LABEL: IS-TARGET>)

IS-ACTIVATE = IS-ENABLE V IS-DISABLE V IS-SIGNAL

IS-DSTORE = (<S-ST-ID: IS-STORAGE>,
<S-ENTRY-LIST: IS-ENTRY-LIST>)

IS-STORAGE = IS-ALLOCATE V IS-FREE
APPENDIX C.

VDL PREDICATES FOR ELEMENTARY OBJECTS IN INTERCOM

<table>
<thead>
<tr>
<th>ALLOCATE</th>
<th>is-allocate</th>
<th>HALT</th>
<th>is-halt</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND</td>
<td>is-and</td>
<td>HEX</td>
<td>is-hex</td>
</tr>
<tr>
<td>BEGIN</td>
<td>is-begin</td>
<td>HIERARCHY</td>
<td>is-hierarchy</td>
</tr>
<tr>
<td>BUFFER</td>
<td>is-buffer</td>
<td>IF</td>
<td>is-if</td>
</tr>
<tr>
<td>BYTE</td>
<td>is-byte</td>
<td>INPUT</td>
<td>is-input</td>
</tr>
<tr>
<td>CALL</td>
<td>is-call</td>
<td>LET</td>
<td>is-let</td>
</tr>
<tr>
<td>CHAR</td>
<td>is-char</td>
<td>NOT</td>
<td>is-not</td>
</tr>
<tr>
<td>DECLARE</td>
<td>is-declare</td>
<td>ON</td>
<td>is-on</td>
</tr>
<tr>
<td>DISABLE</td>
<td>is-disable</td>
<td>OR</td>
<td>is-or</td>
</tr>
<tr>
<td>ELSE</td>
<td>is-else</td>
<td>RETURN</td>
<td>is-return</td>
</tr>
<tr>
<td>ENABLE</td>
<td>is-enable</td>
<td>SIGNAL</td>
<td>is-signal</td>
</tr>
<tr>
<td>END</td>
<td>is-end</td>
<td>WAIT</td>
<td>is-wait</td>
</tr>
<tr>
<td>FREE</td>
<td>is-free</td>
<td>WHILE</td>
<td>is-while</td>
</tr>
<tr>
<td>GO</td>
<td>is-go</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

< is-lsthan  | | | |
> is-grtthan | | | |
$ is-comp  | & is-lgcand |
@ is-adr   | # is-xor |
% is-indirect | <= is-1seq |
= is-assign | >= is-greq |
+ is-plus  | == is-eq |
- is-minus | != is-ne |
* is-multiply | << is-lshft |
/ is-divide | >> is-rshft |
APPENDIX D.
AN ABSTRACT INTERPRETER FOR INTERCOM

I1. "interpret system"
int-system(S) =
   PASS := null;
   int-stmt-list(t);
   t: S-CONTROL(P(S-CTLNUM(S)))
   S-CTLNUM := 1;
   {scan(pc)|12<=pc<=sys-length(S-CONTROL(S))};
   up-ctlnum;
   create-tables(h,1);
   h: S-CONTROL(P(S-CTLNUM(S))}
   int-dcls(S-DCLS-LIST(k))
   k: S-CONTROL(P(S-CTLNUM(S))}

I2. "interpret declarations list"
Int-dcls-list(t) =
   head(t) -> null then
      int-dcls-list(tail(t));
      up-stc;
      int-dcls(head(t))
   TRUE then PASS := null

I3. "head of a tree"
Head(t) =
   first branch of a tree list
   (not further defined)

I4. "tail of a tree"
Tail(t) =
   the subtree of t with
   branch head(t) removed
   (not further defined)
I5. "interpret declarations"
\[ \text{Int-dcls}(t) = \]
\[ \text{int-entry-list}(S\text{-ENTRY-LIST}(t)) \]

I6. "interpret entry list"
\[ \text{Int-entry-list}(t) = \]
\[ \text{head}(t) = \text{null} \text{ then} \]
\[ \text{int-entry-list}(\text{tail}(t)); \]
\[ \text{int-entry}(\text{head}(t)) \]
\[ \text{TRUE then PASS} := \text{null} \]

I7. "interpret entry"
\[ \text{Int-entry}(t) = \]
\[ \text{IS-QUALIF-NAME}(S\text{-NAME}(t)) \text{ then} \]
\[ \{ \text{uptbls}(t) \mid 1 \leq k \leq \text{eval}(S\text{-SUBSCRIPT}(t)) \} \]
\[ \text{TRUE then uptbls}(t) \]

I8. "update environment and attribute tables"
\[ \text{Uptbls}(t) = \]
\[ \text{TRUE then} \]
\[ \text{update-atrtbl}(t); \]
\[ \text{update-envtbl}(t) \]

I9. "create tables"
\[ \text{create tables}(a, z) = \]
\[ \text{PASS} := \text{null}; \]
\[ \{ \text{build}(a, x) \mid z \leq x \leq \text{prog-length}(a) \}; \]
I10. "build label and denotation label tables"
build(a, b) =

IS-LAB-STAT(S)(elem(b))
elem(b): S-CONTROLL(P(S-CTLNUM(S)) . S-STMT(b))
then PASS := null;
gen-lbl-num;
up-stc;
up-date-dnltbl(c, d);
c : S-LABEL(elem(b))
d : S-STC(a)
update-lbltbl(elem(b))

TRUE then PASS := null;
up-stc

I11. "increment control number"
up-ctlnum =

PASS := null;
S-CTLNUM := S-CTLNUM(S) + 1

I12. "increment statement counter"
up-stc =

PASS := null;
S-CCNTROL(P(S-CTLNUM(S)) . S-STC) :=
S-CONTROL(P(S-CTLNUM(S)) . S-STC(S)) + 1

I13. "build control and interrupt tables"
scan(i) =

PASS := null;
create-tables(c, 2);
c : S-CONTROLL(P(i))
update-intrtbl(elem(1));
update-ctltbl(elem(1));
S-STC := 1;
up-ctlnum;
I14. "generate program number"
gen-prog-num=

\[
\text{PASS} := S-\text{NAMES}(S-\text{PROG-NUM}(S)) \\
S-\text{NAMES}(S-\text{PROG-NUM}) := S-\text{NAMES}(S-\text{PROG-NUM}(S)) + 1
\]

I15. "generate identifier number"
gen-id-num=

\[
\text{PASS} := S-\text{NAMES}(S-\text{ID-NUM}(S)) \\
S-\text{NAMES}(S-\text{ID-NUM}) := S-\text{NAMES}(S-\text{ID-NUM}(S)) + 1
\]

I16. "generate label number"
gen-lbl-num=

\[
S-\text{NAMES}(S-\text{LBL-NUM}) := S-\text{NAMES}(S-\text{LBL-NUM}(S)) + 1
\]

I17. "update control table"
update-ctltbl(c)=

\[
\text{PASS} := \text{null} \\
S-\text{CTLTBL} := \mu(S-\text{CTLTBL}(S); \langle \text{pgm}: p \rangle) \\
p: \text{gen-prog-num} \\
\text{pgm}: S-\text{LABEL}(c)
\]

I18. "update interrupt table"
update-intrtbl(c)=

\[
\text{PASS} := \text{null} \\
S-\text{INTRTBL} := \mu(S-\text{INTRTBL}(S); \langle a:b \rangle) \\
a: P(S-\text{PROG-NUM}(S)) \\
b: \mu(0(\langle S-\text{DEVICE}: x \rangle, \langle S-\text{LEVEL}: y \rangle, \langle S-\text{PVAL}: z \rangle) \\
x: \text{eval}(S-\text{EXP1}(c)) \\
y: \text{eval}(S-\text{EXP2}(c)) \\
z: S-\text{PROG-NUM}(S)
\]
I19. "update denotation label table"
update-dnltbl(t) =

\[
\text{PASS} := \text{null} \\
S\text{-DNLTBL} := \mu (S\text{-DNLTBL}(S); <a;b>) \\
a := \mu (S\text{-LBL-NUM}(S)) \\
b := \mu (\langle S\text{-STC}(t) \rangle) \\
x := \mu (\langle S\text{-CTLNUM}(S) \rangle)
\]

I20. "update label table"
update-lbltbl(u) =

\[
\text{PASS} := \text{null} \\
S\text{-LELTBL} := \mu (S\text{-LELTBL}(S); <\text{lbl};m>) \\
\text{lbl} := S\text{-LABEL}(u) \\
m := \text{gen-lbl-num}
\]

I21. "update attribute table"
update-atrtbl(r) =

\[
\text{PASS} := \text{null} \\
S\text{-ATRTBL} := \mu (S\text{-ATRTBL}(S); <a;b>) \\
a := \mu (S\text{-ID-NUM}(S)) \\
b := \mu (\langle S\text{-SIZE};\text{find-size}(S\text{-ATTRIB}(r)) \rangle, \\
\langle S\text{-HIER};\text{find-hier}(S\text{-ATTRIB}(r)) \rangle, \\
\langle S\text{-UNIT};\text{find-unit}(S\text{-ATTRIB}(r)) \rangle)
\]

I22. "find size"
find-size(b) =

\[
\text{IS-BYTE}(S\text{-ATTR-TYPE}(b)) \text{ then} \\
\text{PASS} := \text{eval}(s\text{-exp}(b)) \\
\text{TRUE then} \text{PASS} := \text{null}
\]
I23. "find hierarchy"
find-hier(t) =

IS-HIERARCHY(S-ATTR-TYPE(b)) then
   PASS := eval(s-exp(b))

TRUE then PASS := null

I24. "find unit"
find-unit(b) =

IS-BUFFER(S-ATTR-TYPE(b)) then
   PASS := eval(s-exp(b))

TRUE then PASS := null

I25. "update denotation table"
update-dentbl(id, val) =

S-DENTBL := mu(S-DENTBL(S); <a:val>)
   a := find-id(id)

I26. "update environment table"
update-envtbl(r) =

PASS := null

S-ENVTBL := mu(S-ENVTBL(S); <id,n>)
   id := S-ID(r)
   n := gen-id-num

I27. "determine system length"
sys-length(S) =

PASS := number of programs in the system
      (not further defined)
I28. "determine program length"
prog-length(t) =
    PASS := number of immediate
    branches in tree t
    (not further defined)

I29. "interpret statement list"
int-stmt-list(t) =
    S-CTLNUM(S) > sys-length(S) then PASS := null
    S-STC(t) <= prog-length(t) then
        int-stmt-list(S-CONTROL(P(S-CTLNUM(S))));
        int-stmt(S-CONTROL(P(S-CTLNUM(S)) \ ELEM(S-STC(T)))
    S-STC(t) > prog-length(t) then
        PASS := null

I30. "interpret statement"
int-stmt(t) =
    IS-IAB-STMT(t) then int-stmt(S-UNLAB-STMT(t));
    IS-BLOCK(t) then int-block(t)
    IS-IF-STMT(t) then int-if(t)
    IS-IF-ELSE(t) then int-else(t)
    IS-ASSIGN-STMT(t) then int-assign(t)
    IS-TRANSFER(t) then int-transfer(t)
    IS-INOUT(t) then int-inout(t)
    IS-CTL(t) then int-ctl(t)
    IS-ACTIVATE-STMT(t) then int-activate(t)
    IS-STORE(t) then int-dstore(t)
    IS-LOOP(t) then int-loop(t)
    IS-CONDITIONS(t) then int-conditions(t)
I31. "interpret block"
int-block(t) =

  TRUE then int-stat-list(S-STMT-LIST(t))

I32. "interpret IF-THEN statement"
int-if(t) =

  test(eval(S-EXP(t)))
  then int-stmt(S-STMT(t));
  up-stc
  TRUE then S-CONTROL(P(S-CTLNUM(S)).STC):=
          prog-length(S-STMT(t))+S-STC(t)+1

I33. "interpret IF-THEN-ELSE statement"
int-else(t) =

  test(eval(S-IF-THEN(S-EXP(t))))
  then S-CONTROL(P(S-CTLNUM(S)).STC):=
          prog-length(S-ELSE-CLAUSE(t))+S-STC(t)+1
          int-stmt(S-IF-THEN(S-STMT(t)))
  TRUE then int-stmt(S-ELSE-CLAUSE(S-STMT(t)))

I34. "true/false test"
test(e) =

  odd(e) then PASS:= TRUE
  even(e) then PASS:= FALSE
  (odd and even are not further defined)

I35. "interpret assignment statement"
int-assign(t) =

  TRUE then eval(S-EXP(t))
I36. 

"interpret transfer statement"

\[
\text{int-transfer}(t) =
\]

\[
\begin{align*}
\text{IS-GO}(S-\text{ST-ID}(t)) \text{ then} & \\
& \text{exec-jump}(\text{tgt}, \text{null}) \\
& \quad \text{tgt: find-target}(S-\text{LBLTBL}(S), S-\text{TARGET}(t)); \\
& \quad \text{up-stc}
\end{align*}
\]

\[
\begin{align*}
\text{IS-CALL}(S-\text{ST-ID}(t)) \text{ then} & \\
& \text{exec-jump}(\text{tgt}, t) \\
& \quad \text{tgt: find-target}(S-\text{LBLTBL}(S), S-\text{TARGET}(t)); \\
& \quad \text{up-stc}
\end{align*}
\]

I37. 

"find target in label table"

\[
\text{find-target}(t, id) =
\]

\[
\begin{align*}
\text{tail}(t) = \text{null} \text{ then } \text{PASS} & := \text{null} \\
\text{LBL}.\text{head}(t) = id \text{ then } \text{PASS} & := LBL(\text{head}(t)) \\
\text{TRUE} \text{ then } \text{PASS} & := \text{find-target}(\text{tail}(t), id)
\end{align*}
\]

I38. 

"execute a jump"

\[
\text{exec-jump}(\text{tgt}, t) =
\]

\[
\begin{align*}
\text{t} = \text{null} \text{ then } & \text{move-ctl}(\text{tgt}) \\
\text{t} \neq \text{null} \text{ then } & \text{move-ctl}(\text{tgt}); \\
& \text{stknum}(\text{tgt})
\end{align*}
\]

I39. 

"stack current statement counter and program number"

\[
\text{stknum}(\text{tgt}) =
\]

\[
\begin{align*}
\text{S-CCNTBCL}(\text{F}(S-\text{DNLTBL}(\text{F}(\text{tgt}).S-\text{PRGNUM}(S))).S-\text{STACK}) & := \\
\mu(S-\text{STACK}(S); <S-\text{RTN-STC}; c>, <S-\text{RTN-PRG}; S-\text{CTLNUM}(S)>) \\
c & := \text{S-CONTROL}(\text{F}(S-\text{CTLNUM}(S)).S-\text{STC}(S))
\end{align*}
\]
I40. "move control to target statement"
move-ctl(tgt) =

\[ S-CTL\text{NUM} := S-DNLTBL(\text{H}(tgt).S-PRG\text{NUM}(S)) \]
\[ S-CC\text{CONTROL} (P(S-CTL\text{NUM}(S)).S-SC) := S-DNLTBL(\text{M}(tgt).S-ST\text{NUM}(S)) \]

I41. "interpret input/output statement"
int-incut(t) =

\[ IS-\text{INPUT}(S-ST-ID(t)) \text{ then input}(a,b,c) \]
\[ IS-\text{OUTPUT}(S-ST-ID(t)) \text{ then output}(a,b,c) \]
\[ a: \text{find}\text{-id}(S-\text{IDENT}(S-\text{NAMES}(S-ID(t)))) \]
\[ b: S-\text{IDENT}(S-\text{SCALE}(t)) \]
\[ c: \text{eval}(S-\text{EXP}(t)) \]

(input and output are not further defined)

I42. "interpret control statement"
int-ctl(t) =

\[ IS-\text{WAIT}(S-ST-ID(t)) \text{ then int-ctl}(t) \]
\[ IS-\text{HALT}(S-ST-ID(t)) \text{ then } S-CTL\text{NUM} := \text{sys}\text{-length}(S)+1 \]
\[ IS-\text{RETURN}(S-ST-ID(t)) \text{ then int-return}(t) \]

I43. "interpret return statement"
int-return(t) =

\[ S-\text{CONTROL}(P(a).S-SC) := \text{mu}(S-\text{CONTROL}(P(a).S-SC(S);<b,c>)) \]
\[ a: S-\text{CONTROL}(P(S-CTL\text{NUM}(S)).S-\text{STACK}(S-\text{RTN-PRG}(S))) \]
\[ b: S-\text{CONTROL}(P(a).S-SC) \]
\[ c: S-\text{CONTROL}(P(S-CTL\text{NUM}(S)).S-\text{STACK}(S-\text{RTN-STC}(S))) \]
\[ S-CTL\text{-NUM} := S-\text{CONTROL}(P(S-CTL\text{NUM}(S)) .S-\text{STACK}(S-\text{RTN-PRG}(S))) \]
I44. "interpret activate statement"
\[ \text{int-activate}(t) = \]
\[
\begin{align*}
\text{IS-ENABLE}(S\text{-ST-ID}(t)) & \text{ then } \\
& \text{set-flag}(t, 1) \\
\text{IS-DISABLE}(S\text{-ST-ID}(t)) & \text{ then } \\
& \text{set-flag}(t, C) \\
\text{IS-SIGNAL}(S\text{-ST-ID}(t)) & \text{ then } \\
& \text{sim-intr}(t)
\end{align*}
\]

I45. "set activation flag"
\[ \text{set-flag}(t,f) = \]
\[
\begin{align*}
S\text{-CONTROL}(P(a)\cdot S\text{-ACT}) & := \\
\mu(S\text{-CONTROL}(P(a)\cdot S\text{-ACT}(S)) ; <C:F>) \\
A: & S\text{-DNLTBL}(M(b)\cdot S\text{-PRGNUM}(S)) \\
B: & \text{FIND-TARGET}(T, S\text{-LABEL}(T)) \\
C: & S\text{-CONTROL}(P(a)\cdot S\text{-ACT})
\end{align*}
\]

I46. "simulate an interrupt"
\[ \text{sim-intr}(t) = \]
\[
\begin{align*}
\text{test}(S\text{-CONTROL}(P(a)\cdot S\text{-ACT}(S)) \\
a: & S\text{-DNLTBL}(M(b)\cdot S\text{-PRGNUM}(S)) \\
b: & \text{find-target}(t, S\text{-LABEL}(t)) \\
& \text{then exec-jump}(a, t) \\
\end{align*}
\]
\[ \text{TRUE then up-stc} \]

I47. "interpret storage allocation"
\[ \text{int-dstore}(t) = \]
\[
\begin{align*}
\text{IS-ALLOCATE}(S\text{-ST-ID}(t)) \text{ then } & \text{int-dcls}(t) \\
\text{IS-FREE}(S\text{-ST-ID}(t)) \text{ then } & \text{int-stgentry}(S\text{-ENTRY-LIST}(t))
\end{align*}
\]
I48. "interpret free storage entry list"
int-stgentry(t) =

    head(t) == null then
    int-stgentry(tail(t));
    int-free(head(t))

    TRUE then PASS := null

I49. "interpret FREE statement"
int-free(t) =

    IS-QUALIF-NAME(S-NAME(t)) then
    {deltbls(t) | 1 <= k <= eval(S-SUBSCRIPT(t))}

    TRUE then deltbls(t)

I50. "delete table entries"
deltbls(t) =

    S-DENTBL := mu(S-DENTBL(S); <a: null>)
    a: find-id(S-ID(t))

    S-ENVTBL := mu(S-ENVTBL(S); <id: null>)

I51. "interpret looping statement"
int-loop(t) =

    test(eval(S-EXP(t)))
    then int-loop(t);
    int-block(S-BLOCK(t))

    TRUE then skip-block(t)

I52. "skip past while block"
skip-block(t) =

    S-CONTROL(P(S-CTINUM(S)).S-STC) :=
    prog-length(S-BLOCK(t)) + S-STC(t) + 1
I53. "interpret conditions statement"

\[ \text{int-conditions}(t) = \]

\[ \text{IS-RETURN}(S-BODY(t)) \text{ then} \]
\[ \text{int-return}(t) \]

\[ \text{IS-BLOCK}(S-BODY(t)) \text{ then} \]
\[ \text{int-block}(S-BODY(t)) \]

I54. "find identifier in environment table"

\[ \text{find-id}(id) = \]

\[ \text{tail}(t) = \text{null} \text{ then} \text{PASS} := \text{null} \]

\[ \text{ID}.\text{head}(t) = id \text{ then} \text{PASS} := \text{ID(head}(t)) \]

\[ \text{TRUE} \text{ then} \text{PASS} := \text{find-id(tail(t), id)} \]

\[ t: S-\text{ENVtbl}(S) \]

I55. "expression evaluation"

\[ \text{eval}(t) = \]

\[ \text{IS-BIN}(t) \text{ then} \text{int-bin}(S-\text{RATOR}(t), a, b) \]
\[ a: \text{eval}(S-\text{OPRND1}(t)) \]
\[ b: \text{eval}(S-\text{OPRND2}(t)) \]

\[ \text{IS-UNARY-FACTOR}(t) \text{ then} \text{int-unary}(S-\text{UNARY}(t), a) \]
\[ a: \text{int-factor}(S-\text{FACTOR}(t)) \]

\[ \text{IS-FACTOR}(t) \text{ then} \text{int-factor}(t) \]

I56. "binary operation evaluation"

\[ \text{int-bin}(op, oprnd1, oprnd2) = \]

\[ \text{TRUE} \text{ then} \text{PASS} := \text{apply(oprnd1, op, oprnd2)} \]

I57. "unary operation evaluation"

\[ \text{int-unary}(op, oprnd1) = \]

\[ \text{TRUE} \text{ then} \text{PASS} := \text{apply(oprnd1, op, null)} \]
I58. "interpret factor"
\texttt{int-factor(t) =}
\begin{itemize}
  \item \texttt{IS-FUNCTION(t) then PASS:= int-function(t)}
  \item \texttt{IS-IDENT(t) then PASS:= find-id(t)}
  \item \texttt{IS-INT(t) then PASS:= add-con(t)}
  \item \texttt{IS-EXP(t) then PASS:= eval(t)}
\end{itemize}

I59. "interpret function"
\texttt{int-function(t) =}
\begin{itemize}
  \item \texttt{IS-CHAR-TYPE(t) then add-con(S-CHAR-ARG(t))}
  \item \texttt{IS-HEX-TYPE(t) then add-con(S-HEX-ARG(t))}
\end{itemize}

I60. "add constants to tables"
\texttt{add-con(t) =}
\begin{itemize}
  \item \texttt{S-DENTBL:= mu(S-DENTBL(S);<get-loc:t>)}
\end{itemize}

I61. "get a location for the constant"
\texttt{get-loc=}
\begin{itemize}
  \item \texttt{PASS:= S-NAMES(S-ID-NUH(S));}
  \item \texttt{S-ENVTBL:= mu(S-ENVTBL(S);<dummy:n>)}
  \item \texttt{n: gen-id-num}
\end{itemize}

I62. "apply the operator"
\texttt{apply(a,op,b)=}
\begin{itemize}
  \item \texttt{perform the operation specified by op}
  \item \texttt{on the information at locations}
  \item \texttt{whose addresses are a and b}
  \item \texttt{(not further defined)}
\end{itemize}
APPENDIX E.
FLOW OF CONTROL IN A SIMPLE SYSTEM

START

1

send next character to terminal

NO

message done?

YES

output done?

NO

input done?

YES

attention?

NO

disconnect?

YES

NOTES

start sending to terminal

translate message to ASCII

translate message to ASCII

NOTES

start sending to terminal

translate message to ASCII

translate message to ASCII

NOTES

send next character to terminal

NO

message done?

YES

output done?

NO

input done?

YES

attention?

NO

disconnect?

YES

NOTES

start sending to terminal

translate message to ASCII

translate message to ASCII

NOTES

send next character to terminal

NO

message done?

YES

output done?

NOTES

NOTES

NOTES

NOTES

send next character to terminal

NO

message done?

YES

output done?

NOTES

NOTES

NOTES

NOTES

send next character to terminal

NO

message done?

YES

output done?

NOTES

NOTES

NOTES

NOTES

send next character to terminal

NO

message done?

YES

output done?

NOTES

NOTES

NOTES

NOTES

send next character to terminal

NO

message done?

YES

output done?

NOTES

NOTES

NOTES

NOTES

send next character to terminal

NO

message done?

YES

output done?

NOTES

NOTES

NOTES

NOTES

send next character to terminal

NO

message done?

YES

output done?

NOTES

NOTES

NOTES

NOTES

send next character to terminal

NO

message done?

YES

output done?

NOTES

NOTES

NOTES

NOTES

send next character to terminal

NO

message done?

YES

output done?

NOTES

NOTES

NOTES

NOTES

send next character to terminal

NO

message done?

YES

output done?

NOTES

NOTES

NOTES

NOTES
overflow ?

line delete ?

back space ?

end of message ?

send message to computer

notify terminal of overflow

reset pointers

translate to EBCDIC

buffer the character

increment pointers

back up buffer cursor

1
APPENDIX F

A SAMPLE PROGRAM IN INTERCOM

START;
"THIS UNTESTED PROGRAM IS A CONCEPTUAL MODEL ONLY."

"RECEIVE INPUT INTERRUPT SERVICE ROUTINES"

:INTR11: ON INTERRUPT(1) UNIT(1)
BEGIN; "TEMPORARILY DISABLE INTERRUPTS"
DISABLE INTR12;
DEV=1; NBUF=NBUF1;
IF N1>75 THEN
BEGIN; "OVERFLOW EXISTS"
CALL ERR1;
RETURN;
END;
N=N1; BFADR=@TBUF1;
CALL EDIT;
N1=N; "RESTORE BYTE COUNTER"
 IF ACCEPT==1 THEN
TBUF1(N1-1)=TRTBL1(NBUF);
ENABLE INTR12; "REENABLE INTERRUPTS"
RETURN;
END;

:INTR12: ON INTERRUPT(1) UNIT(2)
BEGIN; "TEMPORARILY DISABLE INTERRUPTS"
DISABLE INTR11;
DEV=2; NBUF=NBUF2;
IF N2>75 THEN
BEGIN; "OVERFLOW EXISTS"
CALL ERR2;
RETURN;
END;
N=N2; EFADR=@TEUF2;
CALL EDIT;
N2=N; "RESTORE BYTE COUNTER"
 IF ACCEPT==1 THEN
TBUF2(N2-1)=TRTBL1(NBUF);
ENABLE INTR11; "REENABLE INTERRUPTS"
RETURN;
END;

"ATTENTION INTERRUPT SERVICE ROUTINE"

:INTR21: ON INTERRUPT(2) UNIT(1)
BEGIN;
TBUF1.4=HEX(01F1F237);
CALL SNDCC1;
RETURN;

SNDCC1: BEGIN;
PR1=0; N1=2;
DISABLE INT12;
BFADR=@TBUF1; N=4;
CALL SEND;
ENABLE INT12;
RETURN;
END;

:INTR22: ON INTERRUPT(2) UNIT(2)
BEGIN;
TBUF2.4=HEX(01F2F237);
CALL SNDCC2;
RETURN;

:SNLCC2: BEGIN;
PR1=0; N2=2;
DISABLE INT11;
BFADR=@TBUF2; N=4;
CALL SEND;
ENABLE INT11;
RETURN;
END;

"DISCONNECT INTERRUPT SERVICE ROUTINE"

:INTR21: ON INTERRUPT(2) UNIT(1)
BEGIN;
TBUF1.4=HEX(01F1F337);
CALL SNDCC1;
RETURN;
END;

:INTR22: ON INTERRUPT(2) UNIT(2)
BEGIN;
TBUF2.4=HEX(01F2F337);
CALL SNDCC2;
RETURN;
END;

"CHARACTER OUTPUT INTERRUPT SERVICE ROUTINES"

:INTR41: ON INTERRUPT(4) UNIT(1)
BEGIN; "IS PRINTING ALLOWED?"
IF PRT1==0 THEN RETURN;
"CHECK FOR END OF MESSAGE"
IF TBUF1(T1)==HEX(37) THEN
BEGIN;
TBUF1.4=HEX(01F1F137);
CALL SNDCC1;
RETURN;
"INCREMENT BUFFER CURSOR"
T1=T1+1;
"OUTPUT NEXT CHARACTER"
OUTPUT UNIT(1) TBUF1(T1);
RETURN;
END;

:INTR42: ON INTERRUPT (4) UNIT (2)
BEGIN; "IS PRINTING ALLOWED?"
IF PRT2==0 THEN RETURN;
"CHECK FOR END OF MESSAGE"
IF TBUF2(T2)==HEX(37) THEN
BEGIN;
TBUF2.4=HEX(01F2F137);
CALL SNDCC2;
RETURN;
END;
"INCREMENT BUFFER CURSOR"
T2=T2+1;
"OUTPUT NEXT CHARACTER"
OUTPUT UNIT(2) TBUF2(T2);
RETURN;
END;

"MESSAGE FROM CENTRAL COMPUTER COMPLETE"
:INTR63: ON INTERRUPT (6) UNIT (3)
BEGIN;
IF NBUF3(1)=1 THEN
BEGIN;
CALL ETOA;
CCBUF1.75=NBUF3;
T1=0;
CALL TYPE1;
RETURN;
END;
IF NBUF3(1)=2 THEN
BEGIN;
CALL ETOA;
CCBUF2.75=NBUF3;
T2=0;
CALL TYPE2;
RETURN;
END;
HALT; "ERROR IN MSG FORMAT"
END;
"THIS IS THE DECLARATIONS PORTION OF THE PROGRAM"
DECLARE NBUF1 BUFFER UNIT(1);
DECLARE NBUF2 BUFFER UNIT(2);
DECLARE NBUF3(75) BUFFER UNIT(3);
DECLARE TBUF1(75), TBUF2(75);
DECLARE CCBUF1(75), CCBUF2(75);
DECLARE N1, N2, T1, T2;
DECLARE N, NBUF, DEV, BFADR, ACCEPT;
DECLARE TRTBL1(128), TRTBL2(256);

"THIS IS THE MAIN BODY OF THE PROGRAM"

"ASCII TO EBCDIC TRANSLATE TABLE"
TRTBL1=HEX(0010203372D2E2F1605250B0C0D0E0F
101112133C3D322618193F2C1D1E1F
4057F7B56C507D4D5D5C4E6B604B41
F0F12F3F4F5F6F7F8F97A5E4C7E6E6F
7CC1C2C3C4C5C6C7C8C9D1D2D3D4D5D6
D7D8D9E2E3E4E5E6E7E8E9EAEOEPEO
14812838485868788991923949596
979899A2A3A4A5A6A7A8A9244F223507);

"EBCDIC TO ASCII TRANSLATE TABLE"
TRTBL2=HEX(0010203372D2E2F1605250B0C0D0E0F
101112133C3D322618193F2C1D1E1F
4057F7B56C507D4D5D5C4E6B604B41
F0F12F3F4F5F6F7F8F97A5E4C7E6E6F
7CC1C2C3C4C5C6C7C8C9D1D2D3D4D5D6
D7D8D9E2E3E4E5E6E7E8E9EAEOEPEO
14812838485868788991923949596
979899A2A3A4A5A6A7A8A9244F223507);

N1=2; N2=2;
WHILE 1
BEGIN;
WAIT;
END;

"EDIT THE INCOMING CHARACTER"
:EDIT: BEGIN;
"CHECK FOR LINE DELETE"
ACCEPT=0;
IF NBUF==HEX(11) THEN
    BEGIN;
        N=0; RETURN;
    END;

"CHECK FOR BACKSPACE"
IF NBUF==HEX(18) THEN
    BEGIN;
        N=N-1; RETURN;
    END;

"CHECK FOR END OF MESSAGE"
IF NBUF==HEX(37) THEN
    BEGIN;
        "SEND TO COMPUTER"
        CALL SEND;
        N=0; RETURN;
    END;

"OK TO RECEIVE CHARACTER"
ACCEPT=1;
N=N+1; "INCREMENT CURSOR"
RETURN;
END;

"SEND MESSAGE TO CENTRAL COMPUTER"
:SEND: BEGIN;
    OUTPUT UNIT(3) %BPADR.N;
    RETURN;
END;

"TELL TERMINAL ABOUT OVERFLOW"
:ERR1: BEGIN;
    T1=0; PRT1=1;
    CCBUF1.15=CHAR(BUFFER OVERFLOW);
    CCBUF1(15)=HEX(37); "EOT"
    CALL TYPE1;
    RETURN;
END;

"OUTPUT FIRST BYTE TO TERMINAL"
:TYPE1: BEGIN;
    IF PRT1==0 THEN RETURN;
    OUTPUT UNIT(1) CCBUF1(T1);
    RETURN;
END;

:ERR2: BEGIN;
    T2=0; PRT2=1;
    CCBUF2.15=CHAR(BUFFER OVERFLOW);
    CCBUF2(15)=HEX(37); "EOT"
    CALL TYPE2;
    RETURN;
END;
TYPE2: BEGIN;
    IF PET2 == 0 THEN RETURN;
    OUTPUT UNIT (2) CCBUF2 (T2);
    RETURN;
END;

FINISH;