

ESTI FILE COPY

ESD-TR-70-151

ESD RECORD COPY

RETURN TO  
SCIENTIFIC & TECHNICAL INFORMATION DIVISION  
(ESTI), BUILDING 1211

ESD ACCESSION LIST

ESTI Call No. 70287

Copy No. 1 of 1 cys.

Semiannual Technical Summary

Graphics

31 May 1970

Prepared for the Advanced Research Projects Agency  
under Electronic Systems Division Contract AF 19(628)-5167 by

**Lincoln Laboratory**

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Lexington, Massachusetts



AD709187

This document has been approved for public release and sale;  
its distribution is unlimited.

MASSACHUSETTS INSTITUTE OF TECHNOLOGY  
LINCOLN LABORATORY

GRAPHICS

SEMIANNUAL TECHNICAL SUMMARY REPORT  
TO THE  
ADVANCED RESEARCH PROJECTS AGENCY

1 DECEMBER 1969 - 31 MAY 1970

ISSUED 26 JUNE 1970

This document has been approved for public release and sale;  
its distribution is unlimited.

LEXINGTON

MASSACHUSETTS

The work reported in this document was performed at Lincoln Laboratory, a center for research operated by Massachusetts Institute of Technology. This work was sponsored by the Advanced Research Projects Agency of the Department of Defense under Air Force Contract AF 19(628)-5167 (ARPA Order 691).

This report may be reproduced to satisfy needs of U.S. Government agencies.

Non-Lincoln Recipients

**PLEASE DO NOT RETURN**

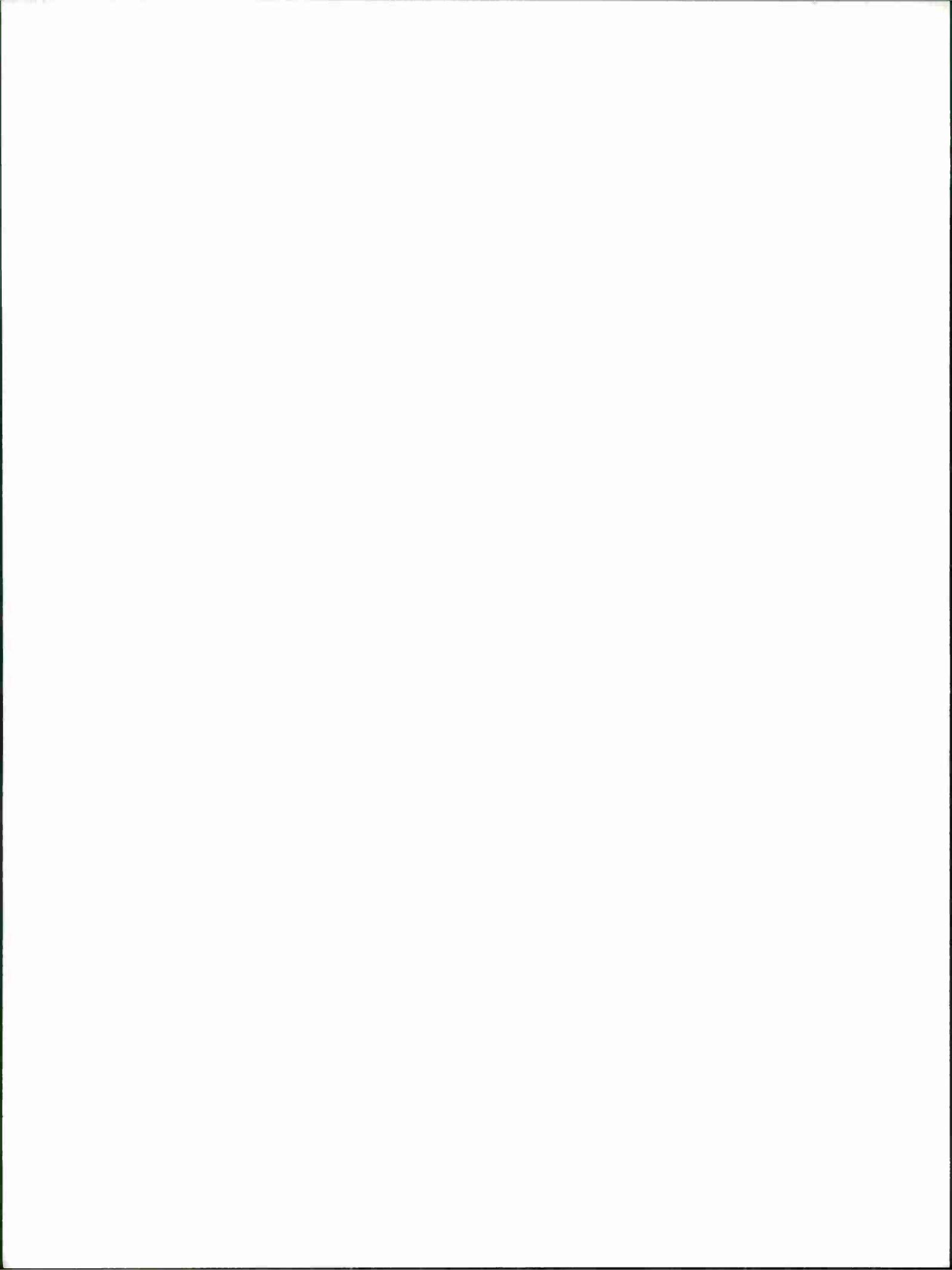
Permission is given to destroy this document  
when it is no longer needed.

## SUMMARY

Software design for the Terminal Support Processor (TSP) system has concentrated on the specification of a language called LIL (for Local Interaction Language). Designed for interpretation by a microprocessor in the TSP system, LIL is a general-purpose language with primitives for manipulating display structures and handling message-oriented input-output. The user specifications for LIL are now available and are presented here in considerable detail.

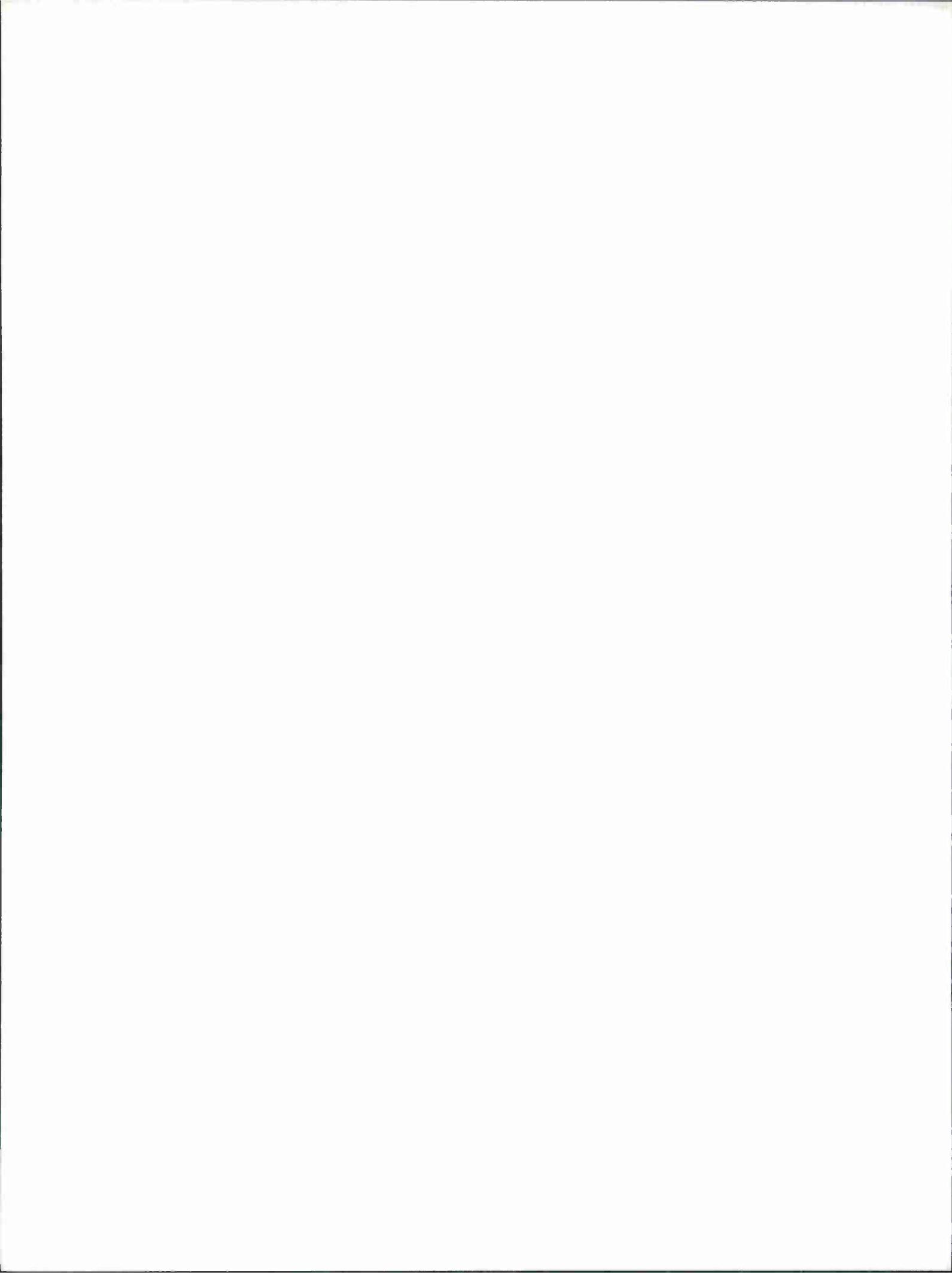
A new mechanism for triggering a user program at interrupt level has been implemented on TX-2. The mechanism uses signals derived from hardware devices which can monitor the state of TX-2 control registers. An experimental interactive program has been written to illustrate one application area for the new facility: software measurement. A new character generator has been installed on TX-2. The storage scope editor on TX-2 has been refined and extended on the basis of user experience. Cursor visibility has been improved by flashing the cursor at a rate of six per second. The Basic Combined Programming Language (BCPL) compiler on TX-2 has been optimized. An overall compilation speed improvement of 374 percent has been achieved in part by making use of the new performance measurement tools now available. An on-line documentation system has been developed for preparing, editing, and presenting system documentation on a variety of output devices. On-line documentation is now available for many parts of the TX-2 system.

Accepted for the Air Force  
Franklin C. Hudson  
Chief, Lincoln Laboratory Office



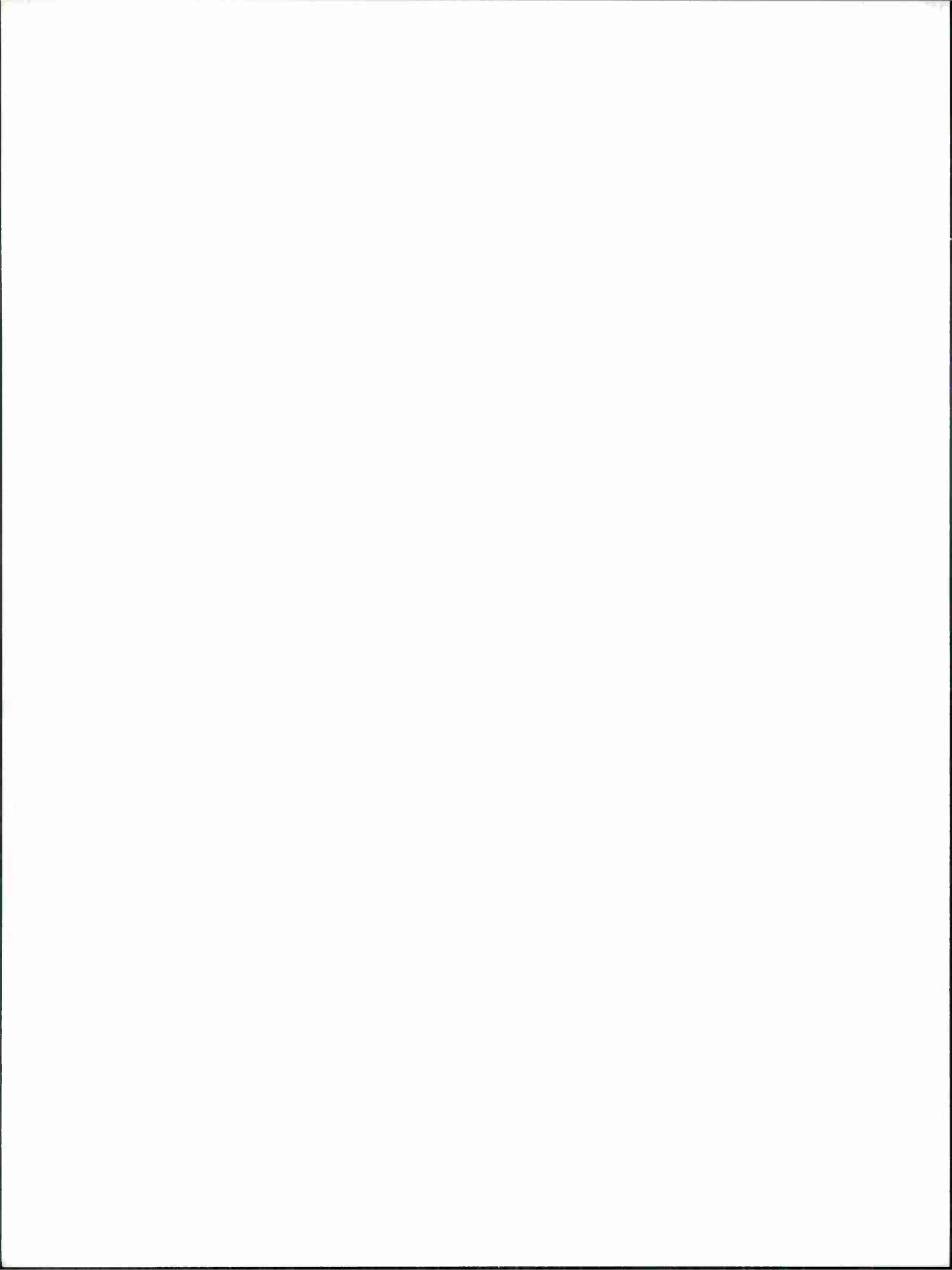
## CONTENTS

Summary	iii
Glossary	vii
I. Terminal Support Processor (TSP) System	1
A. LIL - Display Structure	1
B. LIL - The I/O System	3
C. LIL - General-Purpose Language	5
II. TX-2 System	8
A. Performance Measurement	8
B. Sample Measurements	8
C. Character Generator	9
D. Storage Scope Editor	9
E. BCPL Compiler	11
F. On-Line Documentation	11



## GLOSSARY

ALGOL	A high-level algebraic problem-solving language
APEX	The TX-2 time-sharing system
BCPL	<u>B</u> asic <u>C</u> ombined <u>P</u> rogramming <u>L</u> anguage -- an intermediate-level language for computer programming
LEAP	<u>L</u> anguage for <u>E</u> xpressing <u>A</u> ssociative <u>P</u> rocedures -- an ALGOL-like TX-2 programming language
LIL	<u>L</u> ocal <u>I</u> nteraction <u>L</u> anguage -- a language for use in the TSP system
TAP	<u>T</u> X-2 <u>A</u> ssembly <u>P</u> rogram -- an assembler producing relative binary files
TSP	<u>T</u> erminal <u>S</u> upport <u>P</u> rocessor



# GRAPHICS

## I. TERMINAL SUPPORT PROCESSOR (TSP) SYSTEM

The TSP system is intended to provide local support services to interactive graphics users of the ARPA Computer Network. The design involves an independent processor configuration connected to the network via an IMP (Interface Message Processor) and serving the order of 20 consoles, each consisting of a keyboard, a tablet, and a pair of storage scopes. In addition to basic console support and network control program services, the system allows user program control of all interactions between console input and output, and between TSP and other hosts in the network.

Hardware for the TSP processor has been selected, and delivery is expected in the fourth quarter of 1970. A description of the hardware will be deferred until the procurement procedure is complete.

Software design for the TSP has concentrated on the specification of a language called LIL (for Local Interaction Language), which has been designed for the purpose of specifying the output displays as well as controlling interactions. LIL is a general-purpose language with primitives for manipulating display structures and handling message-oriented input-output; it has been designed for interpretation by a microprocessor in the TSP system. User specifications for LIL are now available and are presented in Secs. A through C below.

### A. LIL - Display Structure

#### 1. Introduction

The design of the display structure has been motivated by the following considerations:

- (a) An interactive graphics user wants fast response to simple requests.
- (b) A minimum of ARPA network communication should be required for graphics.

These considerations lead to the conclusion that the structure describing the display should be kept at the TSP and that the only graphical communication over the network should be requests to change this structure. Since certain interactive requests require no main machine communication, e. g., change the window on the display, they suffer no network delays. Furthermore, an interactive request to delete a part of the structure could be reflected quickly on the scope for the user, even though the message sent to the main machine to update the data structure there may not have been processed, or indeed have arrived.

#### 2. General Description

The display structure consists of subroutines. A subroutine consists of a list of items. Both subroutines and items have user-allocated numeric identifiers. An item contains executable code, whose format will be described later. At the simplest level, an item consists of a call, with parameters of another subroutine. Items are ordered for execution by their item identifiers.

System calls are available to add or delete both subroutines and items from the program structure. These are described later.

Thus, typically, a user would write a set of subroutines which he intends to use as primitive building blocks for his display structure. Calls to such a subroutine might cause different lines to be displayed on the scope, depending on the parameters. For example, in integrated circuit mask layout, such a routine might display a transistor and take as parameters two (x, y) pairs which describe the position and size of the transistor. In a typical application, calls on these subroutines would be added or deleted dynamically under program control.

The display structure subroutines differ only conceptually from those written to handle I/O from, say, the tablet or the network. All subroutines are called in the same manner. Thus, to repaint a picture, the user calls a particular subroutine which (perhaps via subroutine calls) causes execution of all the items necessary to generate the complete picture. Therefore, note that there is only one locus of control for each user. It is his program structure which determines his display hierarchy. This design is possible since we are dealing with storage scopes, which do not have to be refreshed continually.

Subroutines are grouped together into program segments. In general, a user will group together logically related subroutines, or perhaps subroutines which display on only a portion of his display area. Program segments have segment ids which are allocated by the system at segment load time, i. e., when the segment is added to the TSP file system. Program segments are read only. Associated with every program segment is a data segment where the local variables for the program segment are allocated. There is a unique transformation from a program segment id to the id of its associated data segment.

Primitives exist to create and manipulate data segments which are not associated with a particular program segment.

### 3. System Commands for Display Structuring

The following system calls exist for manipulating data and program segments:

- (a) `CREATEDATASEGMENT (<segment name>, <size>, <type>)`  
This call dynamically creates a data segment of length <size> and of type <type>, e. g., READ ONLY. It returns a segment id.  
This call also enters the id and the textual name in the segment name table.
- (b) `CHANGEDATASEGLENGTH (<segment id>, <size>)`  
This call changes the length of <segment id>, to <size>.
- (c) `CREATEPROGRAMSEGMENT (<segment name>)`  
This call creates a program segment. It returns a segment id. The call also creates a data segment whose id is related to the program segment id by a unique one-to-one transformation as mentioned above. The call enters the program segment id and name in the segment name table.
- (d) `DELETESEGMENT (<segment id>)`  
This call deletes the segment, as well as the associated data segment, in the case of program segments.
- (e) `LOADSUBR (<segment id>, <subrid>, <n>)`  
This call creates, in <segment id>, a subroutine called <subrid>, containing no items and having <n> words of local variable storage allocated to it.

- (f) DELETESUBR (<segment id>, <subrid>)  
This call deletes <subrid> in <segment id>.
- (g) CLEARITEMBUFFER  
This call initializes the item buffer, which is where items get built up.
- (h) LOADCALL (<segment id>, <subrid>, <p1>, <p2>, ...)  
This call adds a subroutine call to the item buffer, with the given parameters. The parameters either may be constants, addresses, or values. These are described in Sec. C. Note that the values of the parameters are those at the time the LOADCALL is executed, not at the time <subrid> is executed. In order to get the values of parameters at the time <subrid> is executed, the parameters must be addresses.
- (i) LOADBUFFER (<addr>, <length>)  
This call adds the block of <length> words, starting at <addr>, to the item buffer.
- (j) PUTITEM (<segment id>, <subrid>, <item id>)  
This call puts the contents of the item buffer in <subrid> as <item id>, but does not clear the item buffer.
- (k) DELETEITEM (<segment id>, <subrid>, <item id>)  
This call deletes <item id> from the subroutine.
- (l) REPORTITEM (<segment id>, <subrid>, <item id>, <addr>, <n>)  
This call reports the contents of an item into a buffer of <n> words starting at <addr>.
- (m) SEGID (<segment name>)  
Returns the segment id.
- (n) SEGNAME (<segment id>, <addr>, <n>)  
Returns the segment name into a buffer of <n> words starting at <addr>.

## B. LIL - The I/O System

### 1. Introduction

The TSP user's I/O capability includes two storage scopes, a keyboard, a tablet, and the ARPA network.

In our design, we felt it was important that the type of data transmitted or received should depend as little as possible on the particular physical device employed. Instead, we felt that a canonical representation of the data from a device should be developed. Messages to or from a device would thus be of a particular type.

The I/O system calls in LIL are based on the concept of streams of messages. A stream is a source or sink of messages. A message is one or more 8-bit bytes. What constitutes a message depends upon its type.

Certain devices may be capable of outputting or inputting several types of messages, e. g., a tablet device might produce messages consisting of characters recognized by a character recognizer, or messages of single coordinates or lists of coordinates ("ink points"). Thus, when attaching a device to a stream, the type of message desired must be specified.

Note that several input devices may be attached to the same stream and several output streams to an output device. Thus, messages of different types may be on the same stream. It is only by placing messages on the same stream that a user preserves the time ordering of message transmission.

The defined input devices on the TSP are: (a) Keyboard, (b) Tablet, (c) Network, and (d) Data File.

The defined output devices are: (a) Scopes, (b) Network, and (c) Data File.

## 2. Message Types

(a) Ascii: Two subtypes are defined.

(1) String: A message of type Ascii string consists of a sequence of characters. The number of characters is given in the first byte of the message.

(2) Char: A message of type Ascii char consists of a single character.

(b) Data: A message of type data consists of an arbitrary number of bytes, whether for input or output. The maximum length of a message is 2000 16-bit words. The first two free bytes, after the data type number, of a data message contain the length of the message, in 8-bit bytes.

(c) Tablet: Several subtypes are defined.

(1) Coord: A message of type tablet coord consists of 4 bytes – the first two are an X coordinate, and the second two a Y coordinate.

(2) Inkbuff: A message of type tablet inkbuff consists of two bytes giving the length of the message, followed by tablet coordinates describing the ink points.

(3) Comparator hit: A message of type tablet comparator hit consists of two bytes giving the subroutine id, two bytes giving the item id, and 4 bytes giving the XY position.

There may be more...

## 3. System Commands for I/O

The system calls which are available are:

(a) CREATEINPUT (<stream id>)

This creates an input stream with the given id, i. e., it allocates a buffer area.

(b) CREATEOUTPUT (<stream id>)

This creates an output stream.

(c) ATTACH (<stream id>, <device id>, <type>)

This attaches <device id> to <stream id>.

The <type> is an extension of the <device id> for devices capable of producing several kinds of messages, e. g., the tablet.

(d) DETACH (<stream id>, <device id>, <type>)

This detaches the device from the stream.

(e) CLOSE (<stream id>)

This detaches all devices from the stream. Buffered messages will be lost and further messages not accepted.

- (f) READ (<stream id>, <buffer addr>)  
Read the next message from <stream id> into <buffer addr>. If <stream id> is empty, a special message to that effect will always be in the stream.
- (g) WRITE (<stream id>, <buffer addr>)  
Write the next message into <stream id> from <buffer addr>.
- (h) WAIT  
Wait for a message in any stream, then proceed.

There may be more. . .

### C. LIL - General-Purpose Language

This section contains a description of the general-purpose portion of LIL in which the user codes his display subroutines and his I/O handlers.

The syntactic notation used in this description is basically BNF (Backus Normal Form) with the following extensions: The metalinguistic brackets '<' and '>' may be nested and thus used to group together more than one constituent sequence (which may contain alternatives). An integer subscript may be attached to the metalinguistic bracket '>' and used to specify repetition: if it is the integer  $n$ , then the sequence within the brackets must be repeated at least  $n$  times; if the integer is followed by a minus sign, then the sequence may be repeated at most  $n$  times or it may be absent.

#### 1. TSP Virtual Machine

The virtual machine the TSP user perceives, and in which LIL runs, consists of:

- (a) A segmented address space. Each segment may be up to 2000 words long. The user may define up to  $2^{16}$  segments; in practice, however, he will have a small fraction of this total defined.
- (b) 64 general registers. These registers transcend the segment structure, i. e., they are common to all segments. Arithmetic may be performed on all the registers.
- (c) 8 base registers. These registers also transcend the segment structure. They contain bit patterns which will always be interpreted as segment identifiers.

#### 2. Addresses and Data

Data in LIL consist of 16-bit words. There is no byte addressing. Two types of addresses are defined: register addresses for the 64 general registers, and segment addresses which consist of a base register address and an offset. Unless otherwise specified, either type of address may be used in any operation.

#### 3. Pseudo-Ops

- (a) Syntax:       SEGMENT <name>  
Semantics: This should be the first statement. It defines the name of the segment. A <name> is any sequence of letters and numbers, starting with a letter.

- (b) Syntax: REGISTER {<name> = <constant> <;<name> = <constant>>}\_0  
 Semantics: Use <name> as the name of general register <constant>. This is a data type declaration.
- (c) Syntax: SUBR <id> <n> <par> < , <par>>\_0  
 Semantics: Subroutine called <id> starts here and uses the given parameters as dummy arguments. An <id> is any 16-bit quantity. <n> is the length of the data area associated with this subroutine.
- (d) Syntax: ITEM <id>  
 Semantics: The item called <id> starts here. It is terminated by the next occurrence of the ITEM statement.
- (e) Syntax: VEC {<name> = <value>}  
 Semantics: <name> is a vector of length <value> in this segment, i. e., <name> is a pointer of the 0th word of the vector.

#### 4. Operations

The LIL instruction set is divided into three classes of operations:

- (a) Arithmetic and logical operations
- (b) Control operations
- (c) Miscellaneous operations

##### a. Arithmetic Operations

N2 and N3 will always be considered addresses whose contents are required, unless square brackets [ ] are used which indicate an immediate operand.

- (1) Syntax: N1 ← N2 <op> N3  
 where <op> may be

+ add  
 - subtract  
 \* multiply upper  
 ≅ multiply lower  
 / integer quotient  
 REM integer remainder  
 ^ logical AND  
 v logical OR  
 ⊕ logical EXCLUSIVE OR

- Semantics: (a) In each of the above, N2 may be shifted left or right by a constant, or by the contents of a general register.

For example,

$$A \leftarrow B \ll 3 + C$$

means shift B left by 3 bits.

For example,

$$\text{REGISTER } \{R = 3\}; A \leftarrow B \gg R + C$$

means shift B right by the contents of general register 3.

- (b) A \* (star) next to a name implies indirect. N1, N2 and N3 may all be indirect.

For example,

$$A^* \leftarrow B^* \ll 3 + C$$

means take the contents of the address given by B, shift it left by 3, add C, and store at the address given by the contents of A. Note that the indirect occurs before the shift.

- (c) Every arithmetic statement may have 'S', 'Z' or both appended, implying that the sign or zero condition (or both) of the result is saved in the sign status or zero status bits, respectively.

For example,

$$A \leftarrow B + C, Z$$

means add the contents of A to the contents of B, and store in A. If the result is zero, store 1 in the zero status bit, otherwise, store 0.

- (2) Syntax:  $N1 \leftarrow N2$

Semantics: Assign N2 to N1. Shifting N2, indirection and saving the condition bits is permitted.

b. Control Operations

- (1) Syntax: JUMP <s>, <z> <itemid>

JUMPREL <s>, <z> <n instr> where  
<n instr> is a self-relative address, <s> may be S,  $\bar{S}$  or blank, and <z> may be Z,  $\bar{Z}$  or blank.

Semantics: Jump if the S and Z bits match the given bits; e. g.,

- a. JUMP,  $\bar{Z}$  L means jump to item L if the Z bit is 0, and the S bit is anything.
- b. JUMPREL, -3 means jump unconditional up 3 operations.

- (2) Syntax: SWITCHVIA <addr> <L1>, <L2>, ...

Semantics: Jump to the *i*<sup>th</sup> label in the list, where *i* is the value of <addr>. A label is always a self-relative address.

- (3) Syntax: DO <loopvar>, <incr>, <endvalue>, <label>

Semantics: Compute <loopvar> + <incr>, store in <loopvar>, compare to <endvalue>, and jump to <label> if <loopvar>  $\leq$  <endvalue>.

- (4) Syntax: CALL <subrid> <segid> <p1>... .

Semantics: Call subroutine <id> with parameters <p1>, <p2>... . The return address is stored in locations 0 and 1 of the data area of the called subroutine.

Location 0 contains the calling segment id, location 1 contains the offset.

The values of the parameters are then placed from location 2 on.

- (5) Syntax: RETURN

Semantics: Return from subroutine call using the return address found in locations 0 and 1 of the data area of the subroutine.

- (6) Syntax: FINISH

Semantics: Return to "system."

### c. Miscellaneous Operations

This is whatever is left.

- (1) Syntax:      LOADBR <n>, <addr>  
Semantics: Load base register <n> (<n> = 1, . . . , 7) with the contents of <addr>.
- (2) Syntax:      STOREBR <n>, <addr>  
Semantics: Store the contents of base register <n> in <addr>.
- (3) Syntax:      COMMAND <name> <p1> <p2> . . .  
Semantics: What follows is a system command, e. g., I/O or structuring. The name of the system command is <name>, and its parameters are <p1> <p2> . . .  
Error handling is not yet specified.

## II. TX-2 SYSTEM

### A. Performance Measurement

A new mechanism for triggering a user program at interrupt level has been implemented in the APEX time-sharing system on TX-2: the outputs of the two "sync sources," and the overflow bits on the two event counters. The sync sources are hardware devices for monitoring the state of the TX-2's control registers. Each sync source outputs a pulse each time the specified control register(s) reach(es) the indicated state(s). For example, a sync source may be set up to output a pulse each time the program counter equals 60135, or each time an LDA (load accumulator) instruction is executed, or each time any instruction for the user on console 2 is executed. Each sync source is the input to one of the "event counter" registers, which may be read or written under program control.

An experimental, interactive program has been written to illustrate one application area for the new facility: software measurement. A user may specify a quantity to measure (e. g., program counter, index register contents, working-set size, contents of a specified register, etc.) and several value ranges for the quantity. He may also specify an exact or average frequency for sampling the value of the quantity, based on the state of one event counter (e. g., "every 7000 counts, on the average"). As the application program runs (entirely undisturbed), the monitoring program will sample the indicated quantity at the indicated frequency, and dynamically plot a histogram (on the storage scope) of the number of occurrences of the value in each specified value range.

### B. Sample Measurements

Recently, a flowchart interpreter similar to the one described by W. R. Sutherland<sup>†</sup> has been rewritten in LEAP. Figure 1 is a photograph of a typical flowchart. Individual boxes perform such functions as adding, multiplying, or dividing two numbers, making comparisons, etc. The measurements described here represent interpretations by the flowchart interpreter of this flowchart.

---

<sup>†</sup> W. R. Sutherland, "On-Line Graphical Specification of Computer Procedures," Technical Report 405, Lincoln Laboratory, M.I.T. (23 May 1966), DDC AD-639734.

We began by sampling every 2500 instructions (Fig. 2) in order to get an overview of where the program spends its time. Core was divided into three parts:

- (1) PMATHIO – library code for such things as scope utilities, mathematical routines, text manipulation packages, etc.
- (2) USER – code compiled in-line for the flowchart interpreter.
- (3) LEAP – data structuring routines which manipulate sets, triples, etc.

The overview indicated a much higher percentage of time in the PMATHIO routines than was suspected or understood. The PMATHIO routines were broken down into smaller units and a second run was made (Fig. 3). The constituents which had significant quantities of time were PMATH and RMATH. PMATH contains such pieces as the free storage allocator, block enter/exit events, and several other utilities, thus its usage was not surprising. RMATH, however, contained merely square root, sin/cos, log/exp, and arctan, and its use was most surprising. We broke down RMATH into its pieces and ran once again (Fig. 4), discovering that the time was spent in the square root and log/exp routines. Further investigation revealed that the reason for this high usage was that the interpreter, after interpreting each block, must find the next block to be interpreted. It performs this calculation by traveling down each line from the block and seeing what is near to the end of the line. The distance function used to determine "nearness" is of the form  $\sqrt{x^2 + y^2}$ . It is precisely these calculations which showed up as time in RMATH. The compiler translates  $x^2$  into  $\exp(2 \times \log x)$  which accounts for the time in log and exp routines.

Therefore, we changed the distance function to  $\text{abs}(x) + \text{abs}(y)$ . This trivial modification of the program produced an improvement of 43 percent and took about 2 man-hours to discover and implement [see Fig. 5(a-b)].

### C. Character Generator

The new character generator<sup>†</sup> for the TX-2 graphic display has been installed. This generator utilizes the technique and circuits developed for the TX-2 conic display generator and a read-only memory to store the character stroke data. There are two writing speeds – one for refresh display scopes, and one for storage scopes. The average speed is 25  $\mu\text{sec}$  per character for refresh scope and 300  $\mu\text{sec}$  per character for storage scope. The character writing rate for storage scopes is slower than originally planned. This allows for variations between the storage CRT characteristics as well as for aging. Samples of these character displays are shown in Figs. 6 and 7.

### D. Storage Scope Editor

The storage scope editor is now used in preference to others by roughly half the TX-2 users. The remainder generally claim to have no use for its power. Since the editor has become a standard tool, it has been modified to automatically use the refresh scope for those consoles with no storage scope. Previously reported difficulty in distinguishing the cursor (displayed in "write-thru" mode on the storage scope) has been overcome by a change to the APEX display executive routines which causes information in the write-thru mode to be refreshed no more often than six times per second. The resulting flashing effect draws the eye to the cursor position.

---

<sup>†</sup> Graphics, Semiannual Technical Summary to the Advanced Research Projects Agency, Lincoln Laboratory, M.I.T. (30 November 1969), pp. 6-7, DDC AD-700316.

As a result of experience with the editor, a number of changes have been made in the previously described features and some new features have been added. In the course of this development, the program has grown considerably (from 1650 to 3130 registers).

The commands mentioned in the last semiannual report for bulk text movement have been discarded. In their place, the following commands have been implemented.

- ε – Carves out a section of text from the working file and sends it to be edited recursively. The recursive editor can make a text file out of it or any portions of it, and specify whether the text should be removed from its original position.
- M – Creates a text file out of lines specified by the user.
- A – Appends lines following the cursor to a specified text file.
- t – Inserts a full text file immediately before the cursor. This is specifically used to insert segments of text the user is moving or copying.

To protect users from system crashes, hardware malfunctions, and their own serious mistakes, a drum save feature was added to the storage scope editor. The editor carves out segments from a special drum area assigned to the console, and uses these to save all typing to the editor. The request to write out (save) is given after almost every line, so that usually no more than one line of typing can be lost. By using the text from this area as command input, it is possible to rerun the editing session, in the vast majority of cases recovering fully from the disaster.

By taking command input from a text file instead of the keyboard, it is possible to use the storage scope editor for quite complex changes. To make the use of this facility reasonable, it was necessary to implement commands to switch the command source, and some conditional branching facility. So far, the only conditional branching is done by ignoring the rest of a line when any locate is requested and the string is not found. The following commands have been implemented to facilitate programming:

- j – Stack current command source and start taking commands from a specified text file. This amounts to a recursive subroutine jump.
- h – Abort current command source and start taking commands from a specified text file. This enables exiting a subroutine as well as branch tables.
- q – Test the presence of a string immediately following the cursor.

Since the editor was designed to be indifferent to whether the user moved forward or backwards, it has been found that users quite frequently prefer to move backwards. Consequently, reverse direction locate commands ('λ' and 'n', corresponding to 'L' and 'N') have been installed.

Through monitoring the usage of the storage scope editor (by examining the drum areas where the typing is saved), it has been determined that those commands which take a number as a parameter ('N', 'n', 'X', 'D', 'M' and 'A'), and which allowed the use of a locate string as an afterthought, were actually more useful when used with the locate string option. The monitoring also showed that, because the rules of parameter decoding were sufficiently different for these commands, many users were having difficulty with them. Accordingly, it was decided that they should be thought of as primarily string commands, and the parameter scan was changed to be the same as for the other string commands.

The windowing, or selection of text to display, had come under fire for not showing sufficient context in many cases, and also for not indicating whether any text was above and/or below the window. The windowing routine has been modified to:

- (1) Put tic marks ('^' and 'v') in the left margin to indicate the presence of text above and below the window.
- (2) Show four lines before the cursor when starting a new window.
- (3) Show at least two lines before the cursor whenever the display is repainted.

#### E. BCPL Compiler

Since the LIL compiler is to be written in BCPL, some work has been done to optimize the BCPL compiler. Using the performance measurement facility, we found that most of the compiler's time was spent in the read- and write-character routines. These were carefully hand-coded, speeding up the compiler phase by 37 percent. Further changes have gained an overall speed improvement of 374 percent. The most significant change was the elimination of the first pass of the TAP assembler.

Multiple character streams for the storage scope have been implemented. Experiments using this are in progress to evaluate the notions for possible use in the TSP system.

#### F. On-Line Documentation

An on-line system for APEX user documentation has been developed and is now in regular use. The system consists of two major components – the text, and the supporting hardware.

A simple, but flexible, format for the text was developed which implements outline form through the artifacts of indenting a 'section' by starting it with the desired number of spaces and using one line, however long, per 'section' or 'subsection'.

The supporting software enables one to insert, modify, and retrieve any desired documentation. Output is available via console typewriter, scope (both storage and refresh), or high-speed printer. The output routines break up the long lines into smaller lines to fit the constraints of the output device, and set up the continuations of the lines with the proper indentation. The result is a pleasing and easy to read outline form output for all the output devices.

On-line documentation is now available for all the APEX supervisor calls, for over three quarters of the utility programs, and for miscellaneous other aspects of the APEX environment. Documentation for major systems such as assemblers and compilers is not yet available on-line.

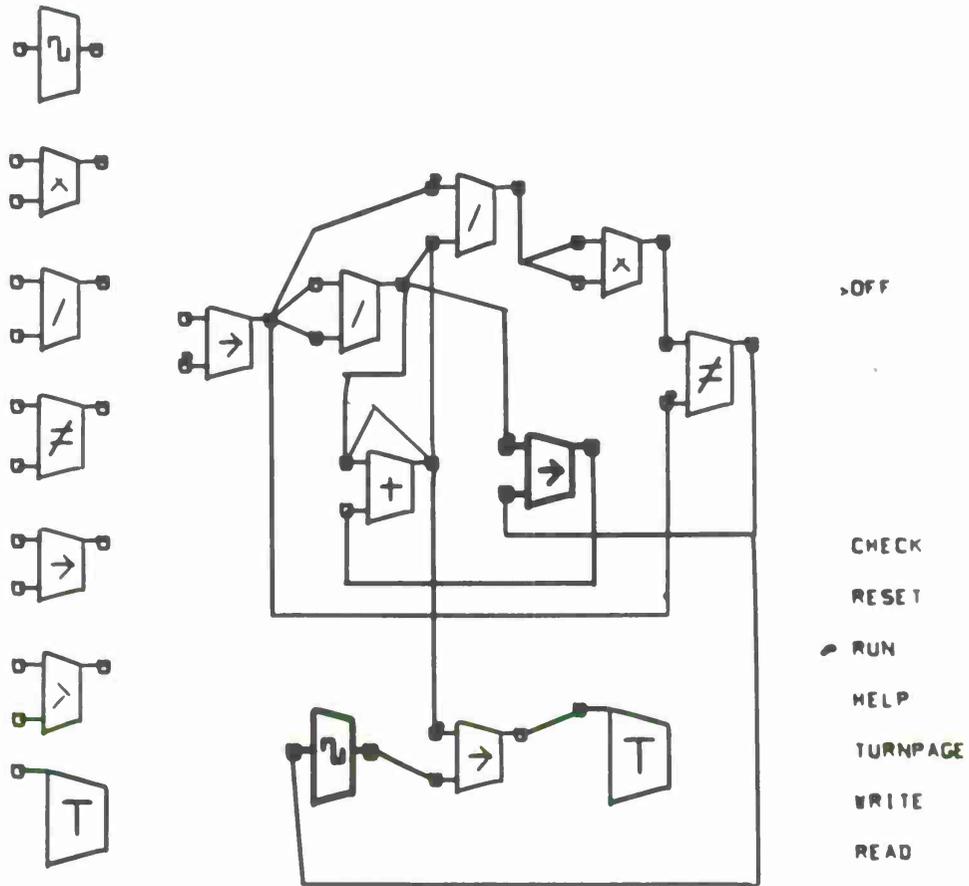


Fig. 1. Typical display produced by flowchart interpreter. This program was sketched in and is now being executed.

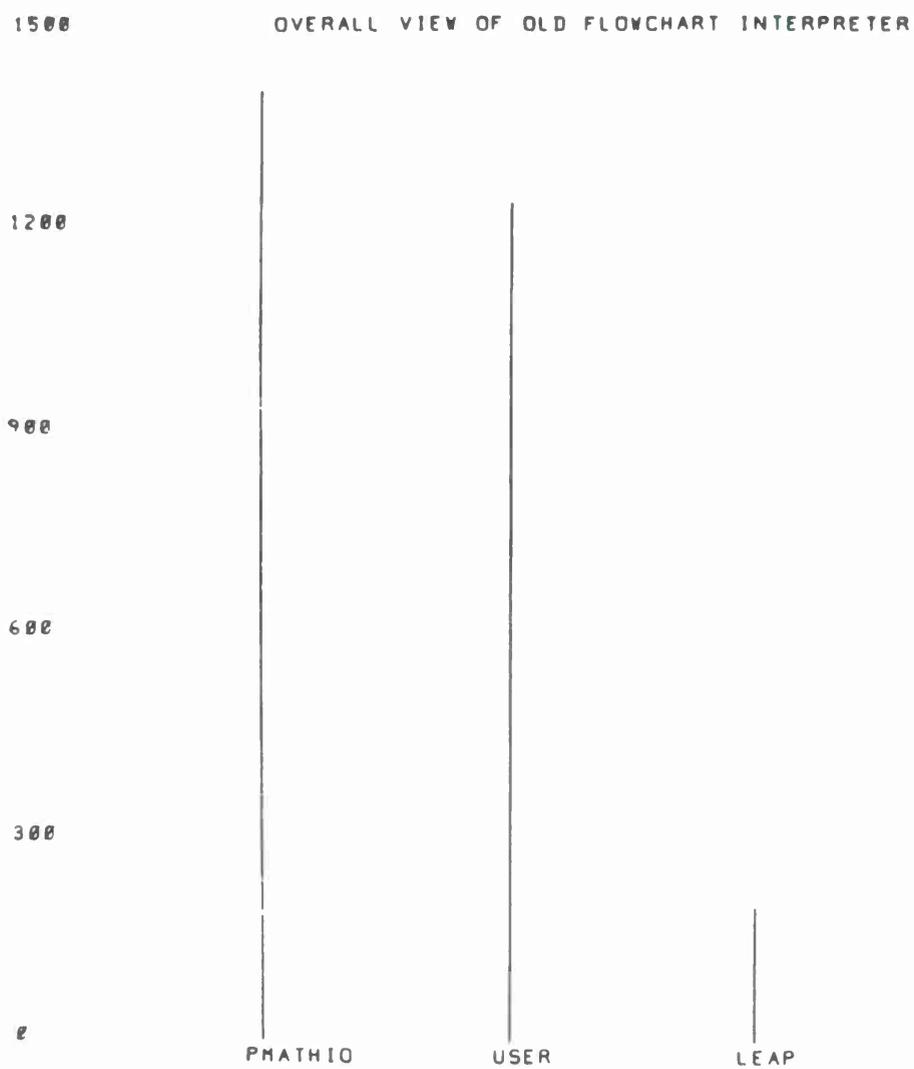


Fig. 2. This graph, photographed directly from the Tektronix 611 storage tube, shows the total number of instructions executed in each of three blocks of core during the interpretation of the flowchart in Fig. 1. Each unit on the vertical axis represents 2500 instructions executed.

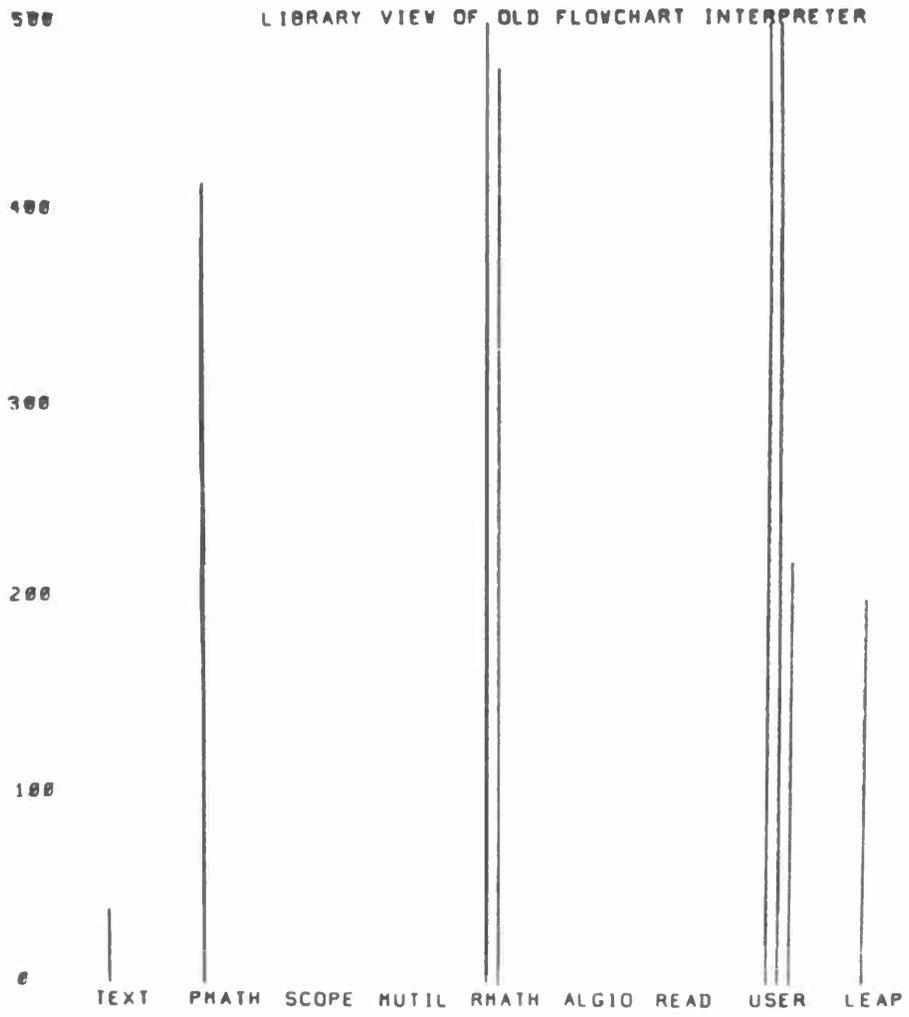


Fig. 3. This graph provides a more detailed breakup of the PMATHIO block in Fig. 2. The vertical scale size is three times larger than in Fig. 2.

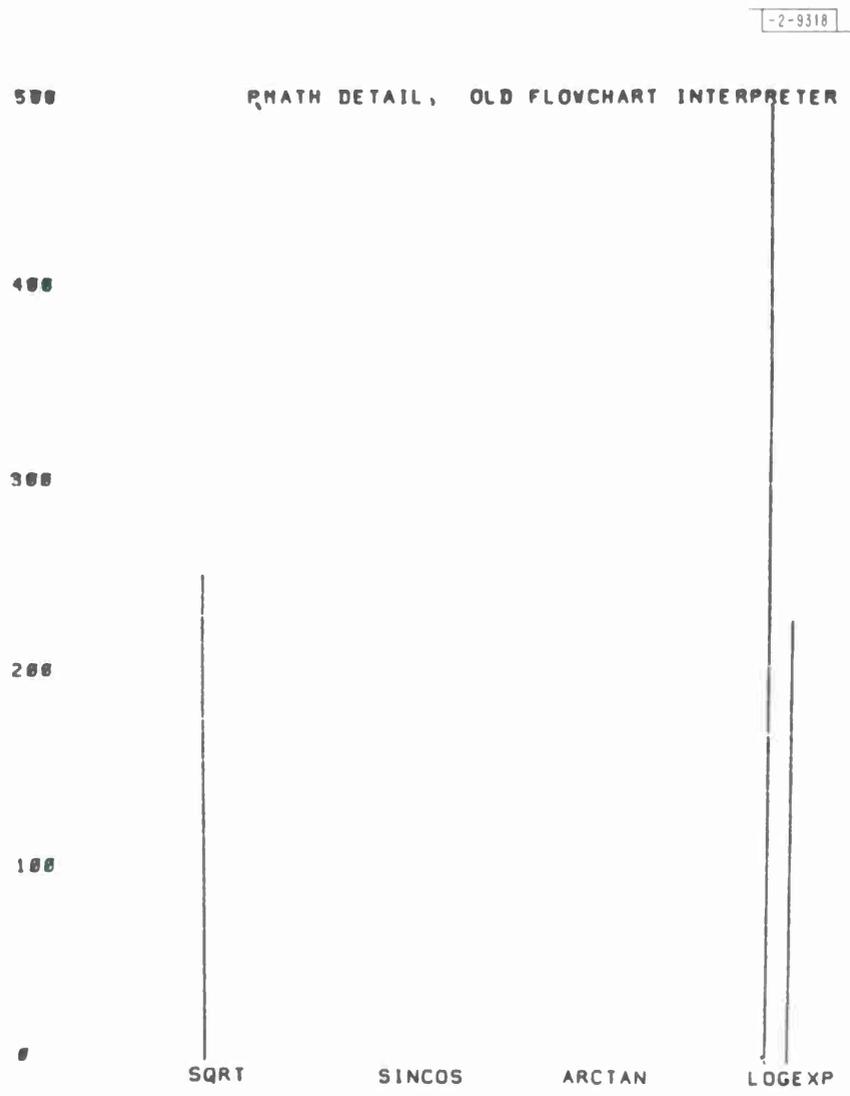
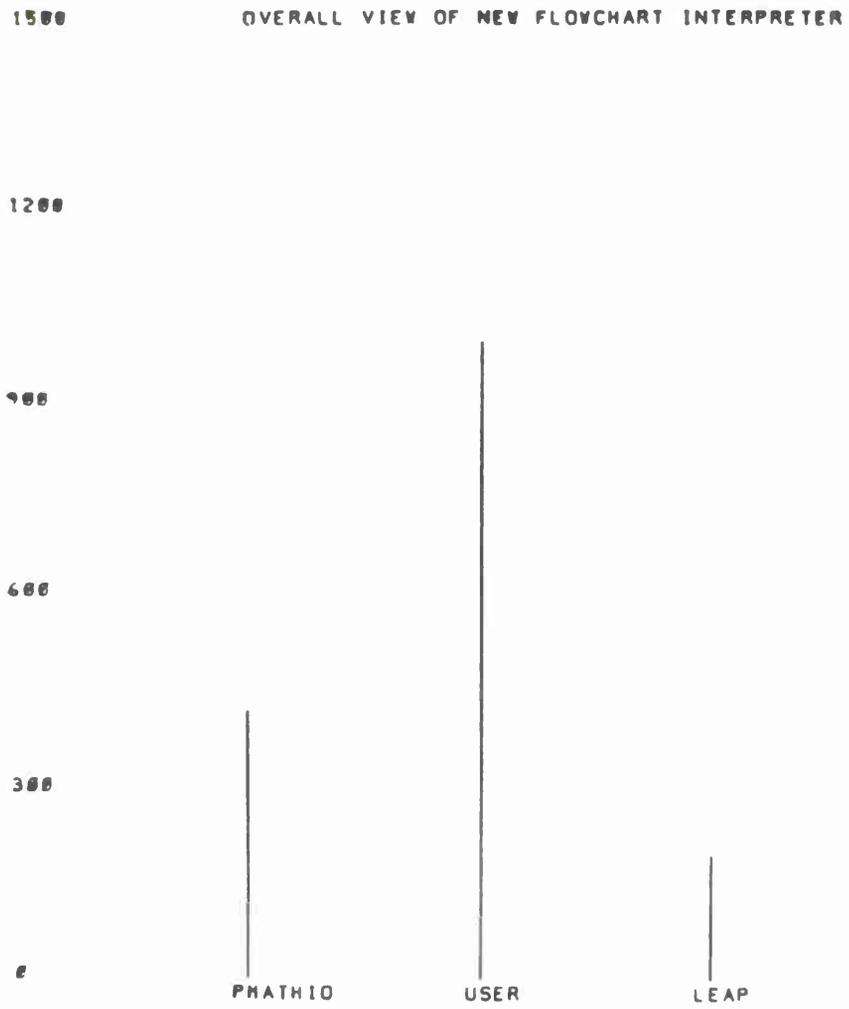
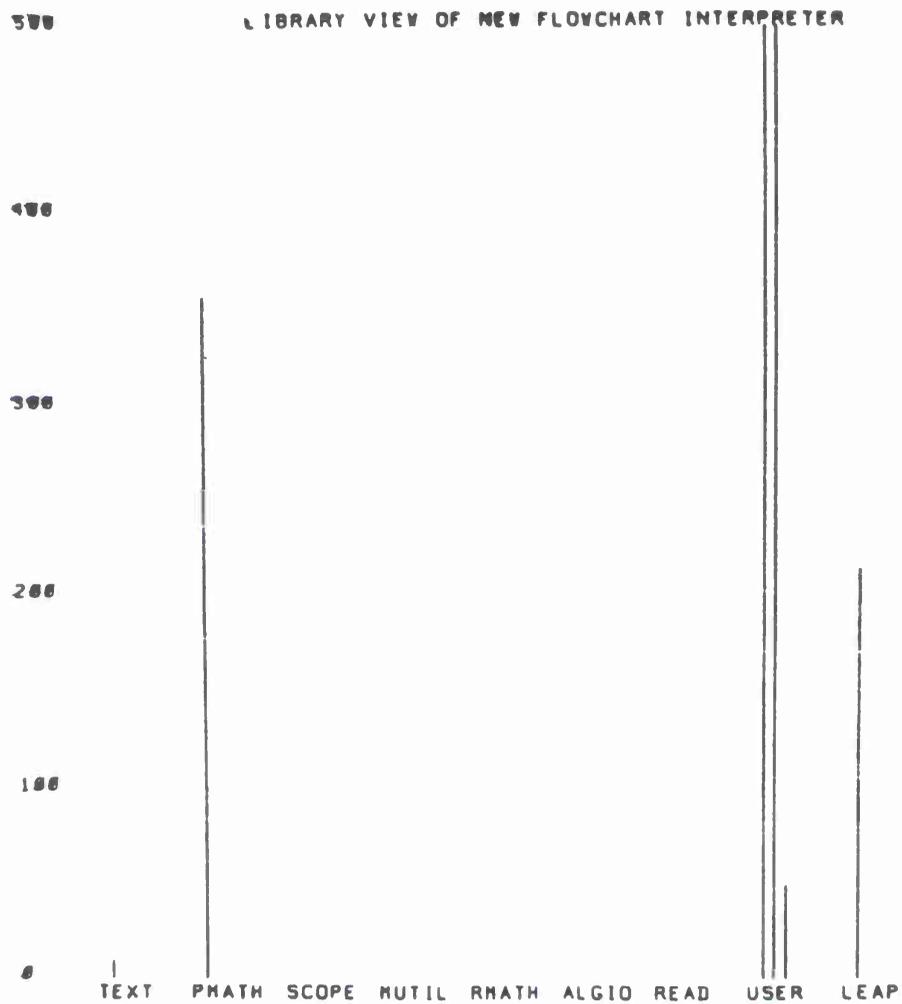


Fig. 4. This detailed view of the RMATH block in Fig. 3 reveals that 34 percent of the total number of instructions were executed in the SQRT and LOG/EXP routines.



(a)

Fig. 5(a-b). These graphs correspond to those of Figs. 2 and 3 after the change in the distance function in the flowchart interpreter.



(b)

Fig. 5. Continued.

PAGE 4 6E

GROUP MEMBERS--		STLINE XS	RO NE BK 13,	STLINE XI	RO BK 2
STLINE XI	FILE			FRI 12 DEC 69	1485 26
STLINE XS	FILE			FRI 12 DEC 69	1485 26
STROKE	FILE			MON 21 APR 69	2128 28
STROKE D	MKS DIR			MON 21 APR 69	2128 15
TARGETTEST	FILE			MON 20 OCT 69	1643 46
TEST3	FILE			MON 21 APR 69	2128 17
TEST6	FILE			MON 21 APR 69	2128 17
TEST9	FILE			MON 21 APR 69	2128 18
TESTREG	FILE			MON 21 APR 69	2128 17
TSPCHARACTERS	TEXT			THU 7 MAY 70	1341 48
TSP 1	TEXT			TUE 19 MAY 70	1324 40
UNSORT	FILE			MON 17 NOV 69	1553 48
V	REF TO 5VITAL			MON 21 APR 69	2128 08
VL	REF TO 5VITLEAP			MON 21 APR 69	2128 08
WJP	TEXT			TUE 19 MAY 70	1325 39
XX	REF TO 5SCRATCH			MON 21 APR 69	2128 08
Z	REF TO 5M5			MON 21 APR 69	2128 08

722

18

Fig. 6. Sample of text as photographed on a 17-inch refresh CRT.

```
PAGE 1          6E          MORE

41E200717      FILE          THU 21 MAY 70 1127.11
6M4DRUM        SCALAR 0      THU 21 MAY 70 1127.11
9LAEP          GROUP 0 AT 6    MON 21 APR 69 2128.07
  GROUP MEMBERS--
9LEAP          GROUP 0 AT 6    MON 21 APR 69 2128.07
  GROUP MEMBERS--9LEAP.SHORTLINE ,
  9LEAP.SPEEDLINE , 9LEAP.RINGS , 9LEAP.TEST9 ,
  9LEAP.TESTREG , 9LEAP.ALLGN , 9LEAP.STLINE
9LEAP.ALLGN    GROUP 0 AT 14    MON 21 APR 69 2128.07
  GROUP MEMBERS--
9LEAP.ALLGN.3D FILE          MON 21 APR 69 2128.15
9LEAP.ALLGN.3L FILE          MON 21 APR 69 2128.19
9LEAP.RINGS    GROUP 0 AT 14    MON 21 APR 69 2128.07
  GROUP MEMBERS--
9LEAP.RINGS.3D FILE          MON 21 APR 69 2128.18
9LEAP.RINGS.3L FILE          MON 21 APR 69 2128.20
9LEAP.SHORTLINE
  GROUP 0 AT 20          MON 21 APR 69 2128.07
  GROUP MEMBERS--
9LEAP.SHORTLINE.3D
  FILE          MON 21 APR 69 2128.18
9LEAP.SHORTLINE.3L
  FILE          MON 21 APR 69 2128.19
9LEAP.SPEEDLINE
  GROUP 0 AT 20          MON 21 APR 69 2128.07
  GROUP MEMBERS--
9LEAP.SPEEDLINE.3D
  FILE          MON 21 APR 69 2128.18
9LEAP.SPEEDLINE.3L
  FILE          MON 21 APR 69 2128.19
```

Fig. 7. Sample of text on a Tektronix 611 storage display unit.

DOCUMENT CONTROL DATA - R&D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY (Corporate author) Lincoln Laboratory, M.I.T.		2a. REPORT SECURITY CLASSIFICATION Unclassified	
		2b. GROUP None	
3. REPORT TITLE Graphics, Semiannual Technical Summary Report to the Advanced Research Projects Agency			
4. DESCRIPTIVE NOTES (Type of report and inclusive dates) Semiannual Technical Summary - 1 December 1969 through 31 May 1970			
5. AUTHOR(S) (Last name, first name, initial) Forgie, James W.			
6. REPORT DATE 31 May 1970		7a. TOTAL NO. OF PAGES 28	7b. NO. OF REFS 2
8a. CONTRACT OR GRANT NO. AF 19(628)-5167		9a. ORIGINATOR'S REPORT NUMBER(S) Semiannual Technical Summary for 31 May 1970	
b. PROJECT NO. ARPA Order 691		9b. OTHER REPORT NO(S) (Any other numbers that may be assigned this report) ESD-TR-70-151	
c.			
d.			
10. AVAILABILITY/LIMITATION NOTICES This document has been approved for public release and sale; its distribution is unlimited.			
11. SUPPLEMENTARY NOTES None		12. SPONSORING MILITARY ACTIVITY Advanced Research Projects Agency, Department of Defense	
13. ABSTRACT <p>Software design for the Terminal Support Processor (TSP) system has concentrated on the specification of a language called LIL (for Local Interaction Language). Designed for interpretation by a micro-processor in the TSP system, LIL is a general-purpose language with primitives for manipulating display structures and handling message-oriented input-output. The user specifications for LIL are now available and are presented here in considerable detail.</p> <p>A new mechanism for triggering a user program at interrupt level has been implemented on TX-2. The mechanism uses signals derived from hardware devices which can monitor the state of TX-2 control registers. An experimental interactive program has been written to illustrate one application area for the new facility: software measurement. A new character generator has been installed on TX-2. The storage scope editor on TX-2 has been refined and extended on the basis of user experience. Cursor visibility has been improved by flashing the cursor at a rate of six per second. The Basic Combined Programming Language (BCPL) compiler on TX-2 has been optimized. An overall compilation speed improvement of 374 percent has been achieved in part by making use of the new performance measurement tools now available. An on-line documentation system has been developed for preparing, editing, and presenting system documentation on a variety of output devices. On-line documentation is now available for many parts of the TX-2 system.</p>			
14. KEY WORDS  graphical communication                      time sharing                      display systems TX-2    man-machine                      programming languages			

