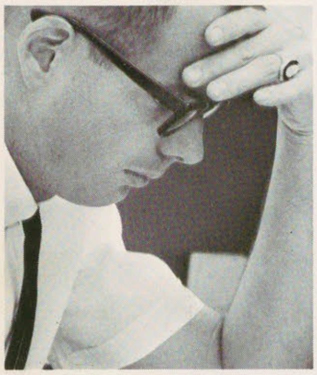
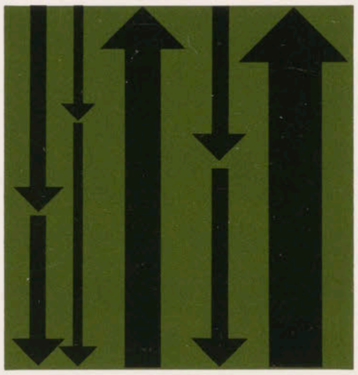
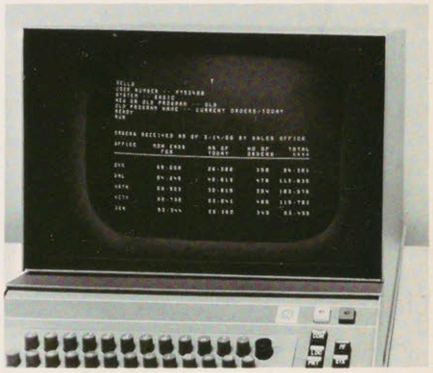
1/2 Crocks c.1960s 1/2

DAYANOY 760

The General Electric DATANET-760 Keyboard/Display







GENERAL ELECTRIC

A New Man-Computer Relationship

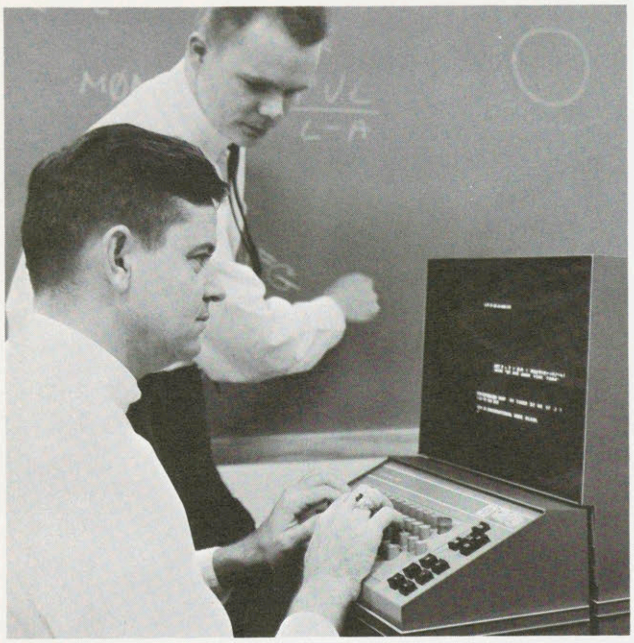
Time-sharing, direct access, real time—these are the buzz-words in data processing today. These techniques are exciting and they are revolutionary—because they let the user talk directly to the computer.

Implicit in this new relationship is the requirement for a device—called a "terminal"—through which man can converse with the computer using symbols familiar to him.

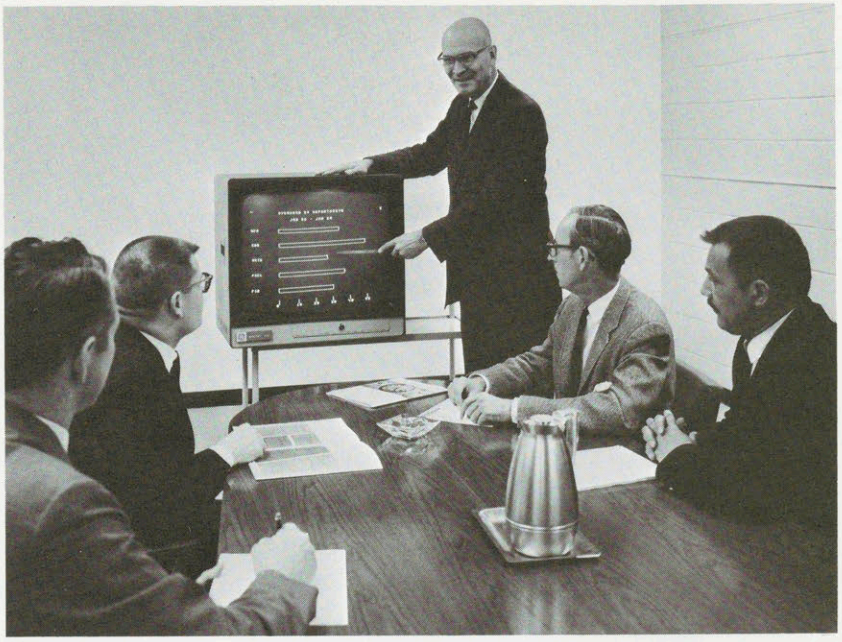
The General Electric DATANET-760* is a Keyboard/Display terminal system which allows the user to talk to the computer in plain English from a distance of a few feet or thousands of miles. With the DATANET-760 at your fingertips you can solve problems, call-out and update computer stored data and, in effect, maintain an efficient and profitable control of every facet of a large and diversified organization.

Virtually anyone can operate the DATANET-760 Keyboard/Display terminal. No previous computer experience is required. General Electric time-sharing computers can talk to you in several easy-to-learn, easy-to-use languages like BASIC (developed at Dartmouth College) and FORTRAN.

**DATANET, Registered Trademark of the General Electric Company*

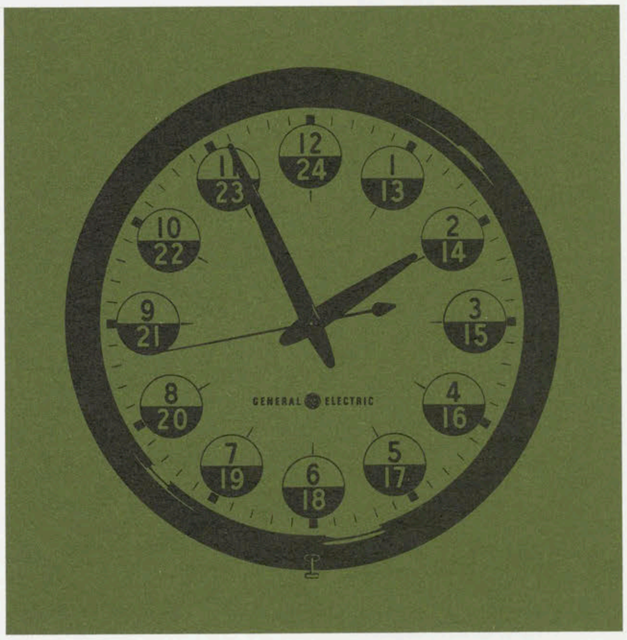


Scientific / Engineering

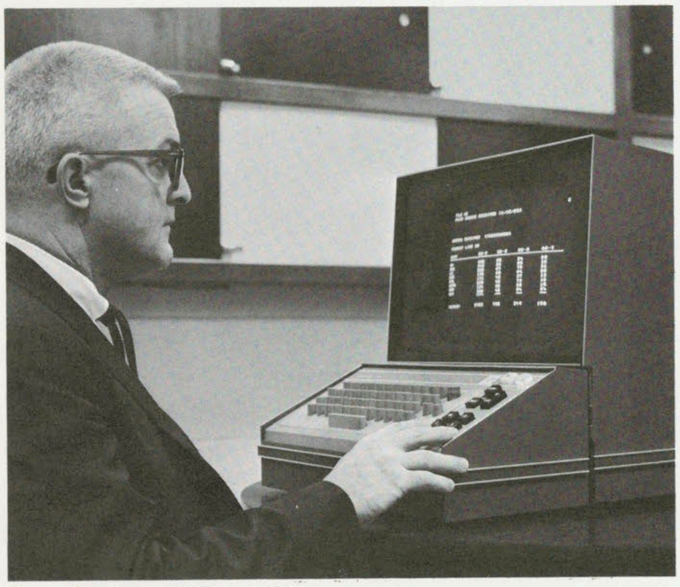


Management Information

Inventory Control



Time-Sharing



Order Inquiry and Processing

Customer Service

Applications

Management

The DATANET-760 provides instantaneous access to computer manipulated information to allow timely, effective management decisions and more profitable operational control. A multitude of complex data—such as project status, sales analysis, customer profile and current profit and loss statements—is available in an instant. *Proved techniques enable selective information to be kept secure from those without the need-to-know.*

Engineering/Scientific

Users in industrial, educational and government research labs no longer need wait for card punching, verifying and human handling delays for solutions to computer-solved problems. The DATANET-760 allows conversation *Directly* with a time-shared computer for real-time problem solving. Problems can be composed on the display screen in the off-line mode—which allows editing and correction before transmission—and then sent to the computer in a single burst. "Scratch pad" calculations can be worked on the screen directly below the primary problem.

Finance

Not only does the DATANET-760 provide direct access to financial and statistical data, it also is a medium through which that information can be kept current. Computer stored files can be updated remotely, at the information source, as new data becomes available. The DATANET-760's formatting and editing capabilities make updating and verification a simple, quick-turnaround operation. Time, manpower and paperwork is conserved.

Marketing

With the DATANET-760, a marketing manager has instant access to such information as shelf inventory status, salesman's activity reports, and customer files. Sales analysis—by territory, product and salesman—can be accomplished in a moment from the marketer's desk. Significant data such as forecasts, effects of price variations on sales/net profit, etc., can be developed with computer assistance and displayed for decision conference or executive meetings via TV monitor or video projection. In addition, the DATANET-760 is perfectly suited for rapid turnaround order entry systems—allowing complete processing of an order, from stock check to remote print-out of shipping documents, to be handled in minutes.

Manufacturing

The DATANET-760 enables manufacturing management to maintain up-to-the-minute control over inventory, material procurement, manpower requirements, production scheduling and expediting. Simply by typing out a request on the keyboard of the DATANET-760 such information as complete vendor performance data, or the record of rejected items can be displayed instantaneously. Decisions can be made and action taken on the basis of current, accurate information.

Personnel

Under computer program control, a personnel manager can have a listing of employees and applicants with a needed skill through the DATANET-760 simply by typing in a description of the desired skill. Complete personnel records, including work history, absentee record, vacation and sick-leave time used/accrued, health insurance claims, etc.,—are available in an instant. Personnel can give managers immediate information about employees and applicants to allow most effective use of manpower.

Customer Service

Queries about charges, account status and often customer complaints are received in great numbers daily by utilities, banks and consumer credit operations. With the DATANET-760, complete customer records can be instantaneously retrieved as customer service personnel are conversing with the customer over the phone or over the counter. Much search time is saved and customers are spared irritating delays. In utility applications, requests for new service can be handled by a "new business" clerk and the new file input by her directly to the computer, via the DATANET-760. A "hook-up" order could be simultaneously printed out remotely at the utility service unit closest to the new customer.

Unique Capabilities of the DATANET-760

Complete Access to Every Character Position on the Display Page

Not only does the operator have complete access to every position on the display, but the DATANET-760 also allows the computer complete access as well. Changes or entries can be made on any line and in any character position without disturbing or rewriting data already displayed on the same line. This is a significant feature, and is especially important in file/form updating applications.

Exceptional Page Display Capacity

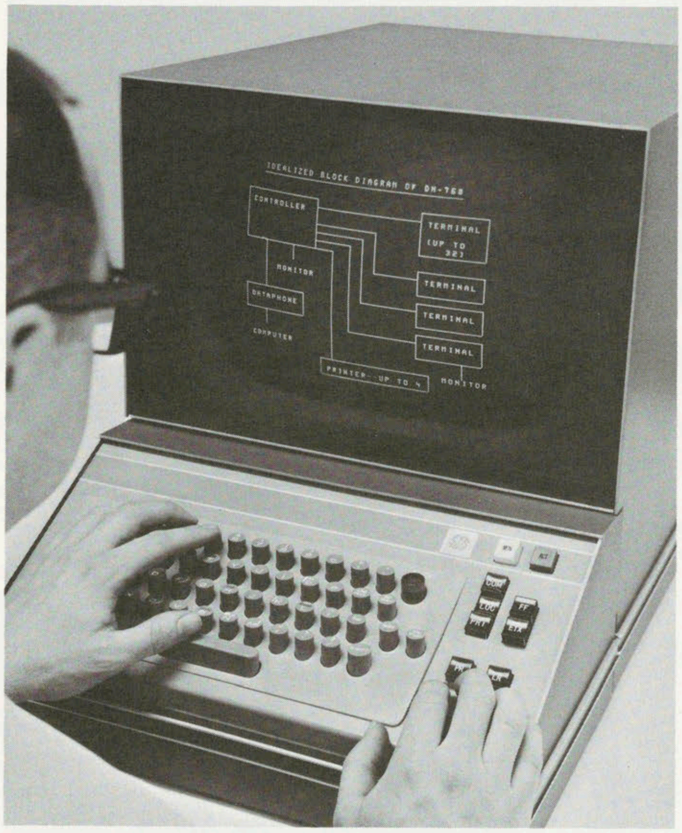
A DATANET-760 terminal may display 1196 symbols in a page array of 26 lines by 46 characters on its 14-inch TV display screen. Because of the modular design of the DATANET-760, terminals may be assigned a full display page, or segments of 4, 8 or 16 textual lines by 46 characters. This allows the most efficient utilization of the system in relation to the functional EDP responsibility assigned each terminal location.

Line Drawing Capability

Special symbols, standard on every DATANET-760 keyboard, allow the operator to draw continuous vertical and horizontal lines on the display screen, as well as segmented diagonal lines. This allows simple charts, diagrams, forms and tables to be drawn on the screen and stored in computer memory.

Standard Television Monitor Compatibility

The DATANET-760 employs a standard 525-line TV display. This standard video compatibility lets you couple many standard, industrial quality television monitors directly to the DATANET-760. This allows the sharing of information, management monitoring of input data, and simple conference hook-ups. Monitors may be co-located with the keyboard display terminal or implaced considerable distances from the terminal. *Status Monitoring*—



the standard TV compatibility also gives the user a very economical status monitoring capability. A terminal memory segment may be assigned a status monitoring function and be addressed and updated by the computer. One or more standard TV monitors may be coupled to that memory segment to provide status "monitor boards" at appropriate locations. *Video Projection* —the data display can be projected for large screen viewing with the use of standard video projection equipment.

Automatic Tab

A tab key and tab stop function let you quickly enter data in computer stored formats. The tab function has a scanning action that lets you move automatically from a position on a particular line to another tab-stop which may be located several lines below.

Error Control and Recovery

A very important feature of the DATANET-760 is its error control capability. If the computer receives a message containing an error it causes the DATANET-760 controller to automatically retransmit the message. If a predetermined number of retransmissions does not result in correction of the error, the computer automatically sends a message notifying the operator that manual recovery is necessary. If an error is detected by the DATANET-760 in a computer generated message, a blinking symbol is displayed on the display screen. This signals the operator that the message is in error and that an automatic retransmission request has been made by the DATANET-760.

Flashing Message

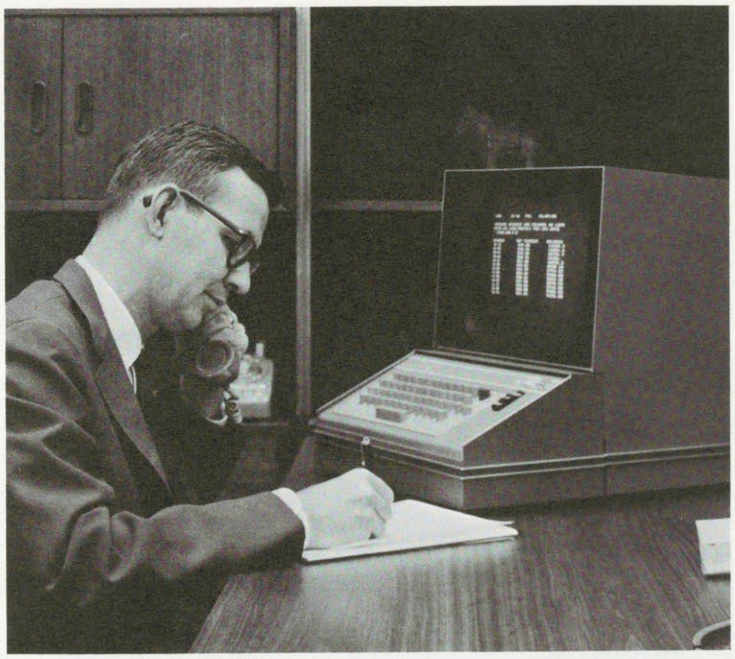
Any single character or field of characters can be displayed in a flashing, or blinking, mode to call special attention to certain information to the operator or viewer. This feature, under program control, is very useful in status reporting or, in file maintenance, to indicate the field of information to be updated.

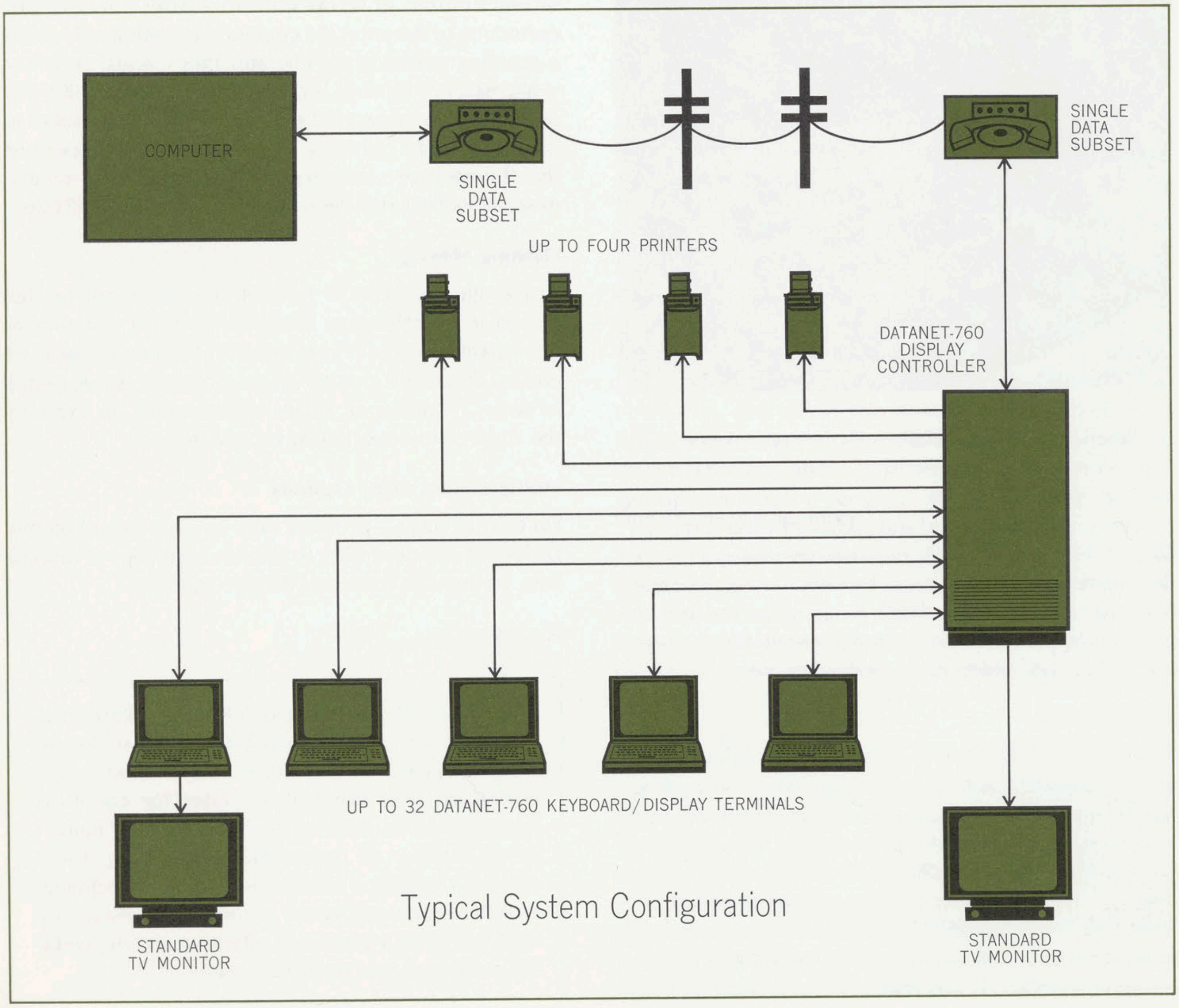
Multiple Hard Copy Options

Up to four page—printers may be connected to the DATANET-760 Controller to provide hard copy of display data on request from any display terminal.

Reliability

The DATANET-760 employs 99% monolithic integrated circuitry for high reliability. General Electric Keyboard/Display sub-systems have been transported all over the United States for countless demonstrations. Despite the "bumps" and "bangs" of all manner of transportation handling, these systems performed admirably. Special attention to realistic communications requirements make the DATANET-760 a genuinely effective and dependable remote access terminal system.





Typical System Configuration

Privacy of Data

In a time-sharing, multiprogramming computer system, it is absolutely essential that there be proven, reliable methods to prevent the accessing of confidential information by uncleared or unqualified individuals. General Electric time-sharing computers have proved techniques that do just that...with just a simple typed statement.

In addition to complete privacy, you can make a file accessible to specific persons and control the condition of their access. With a simple statement, you can make the file available only for reading, only for execution or for full access (read, write or execute).

A Flexible System

The DATANET-760 design is based on highly flexible modular concepts, allowing an information Subsystem to be tailored to individual requirements. Furthermore, a user is not forever committed to an initial configuration. Terminals may be added, their character display capacity increased or decreased, and they may be located and relocated up to 1,000 feet from the central Controller. Post-installation configuration changes or addition of modular options are simple, inexpensive in-field operations.

A typical configuration might consist of up to 32 DATANET-760 Keyboard/Display remote terminals communicating simultaneously through a DATANET-760 Display Controller with a distant computer—via a *single* telephone data communications subset. Additionally, standard TV monitors can be connected to either the Keyboard/Display terminal or the Display Controller.

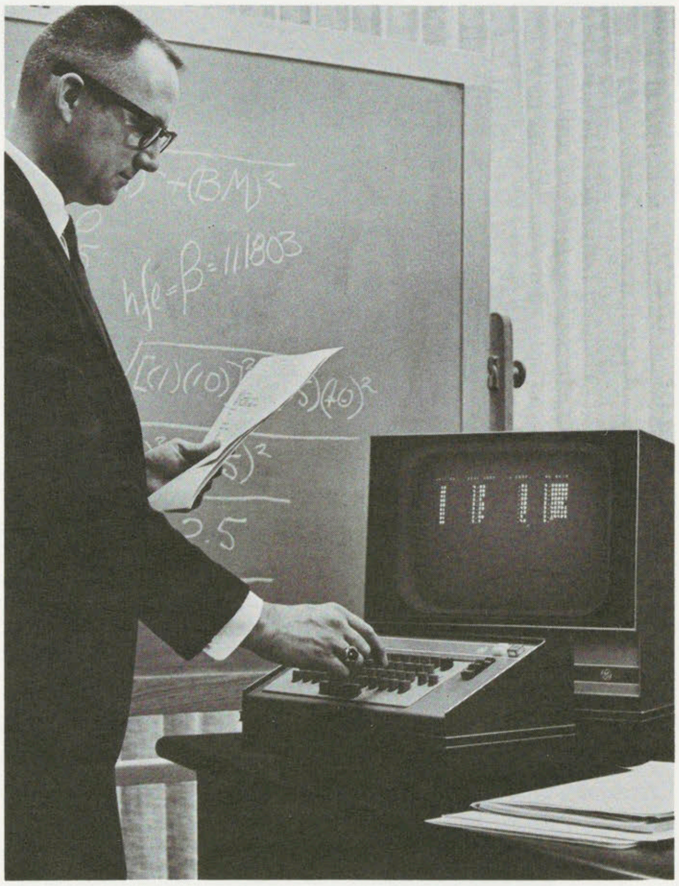
Up to four printers may be connected to the Display Controller to provide simultaneous printout of data as required from any of the Keyboard/Display terminals.

Description of Subsystem

Keyboard/Display Terminal

The DATANET-760 remote terminal consists of a television display module and a keyboard module. The keyboard and display modules may be co-located or separated by several feet. The display module contains a 14-inch cathode ray tube and supporting electronics. Brightness, contrast and other video display controls are easily accessible by the operator. The highly-reliable keyboard has a standard typewriter layout of alphanumeric symbols plus special control keys for drawing lines, transmitting data, etc.

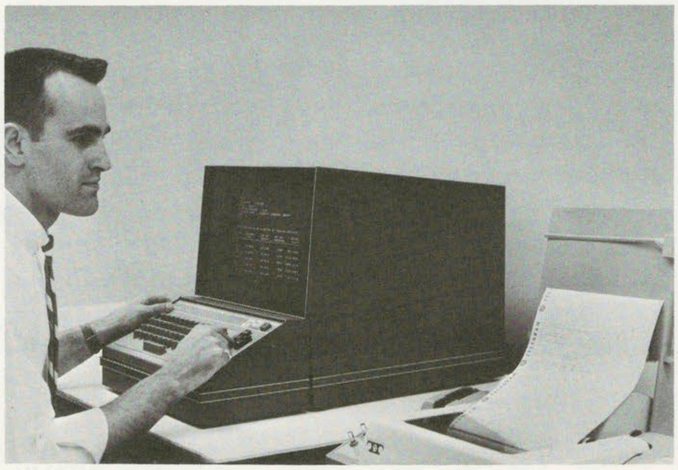
Because of the modular design of the DATANET-760, terminals may be assigned a full display page of 26 textual lines, or segments of 4, 8 or 16 lines by 46 characters.



Display Controller

The Display Controller services up to 32 DATANET-760 remote display terminals, allowing them to communicate simultaneously with the computer by direct connection or through a *single* telephone data communications subset.

The Controller cabinet contains the Basic Display Controller, buffer memories for the Keyboard/Display terminals, and optional Data Line Controller and Page Print Controller modules. Up to four buffer memories—called Terminal Memory Units—may be installed in the Display Controller cabinet. Each Terminal Memory Unit may service up to eight simultaneous access Keyboard/Display terminals. The design is modular, allowing optional modules to be added simply by "plugging them in". Keyboard/Display terminals may be located up to 1000 feet from the Display Controller.



Printer

Any printer comparable to a Teletype Model 33 or 35 Read-Only unit may be interfaced directly to the DATANET-760 or connected remotely through telephone data communications subsets. As many as four printers may be connected to one Display Controller simultaneously, allowing printout of display data from any of the Keyboard/Display terminals.

Specifications Summary

Presentation—page array of symbols (26 textual lines by 46 characters=1196 characters).

Character Repertoire—total of 64 symbols: including 26 alphabetics, 10 numerics and 28 special symbols.

Data Communications Rates—1200 or 2400 bits per second.

Self-Contained Processing and Storage—for off-line composition, editing and correction.

TV Type Display—allows use of standard TV monitors, and TV video projection and distribution equipment. High brightness in office light ambients.

Viewing Surface—14″ rectangular cathode ray tube. Format is approximately 7 x 9.3 inches (on 14″ CRT).

Symbol Matrix—7 x 10 TV lines.

Keyboard Input—standard typewriter key arrangement. Operator has complete access to every character position on the display page, as does the computer.

Character Code—**ASCII** (American Standard Code for Information Interchange).

Transmission Control—allows selection of any portion of display page for transmission to the computer.

Dimensions—Keyboard/Display Terminal: height, 17″; width, 16″; depth, 27″.
Display Controller: height, 62″; width, 27 1/4″; depth, 27 1/4″.

Power Consumption—Keyboard/Display Terminal—200 watts.
Display Controller—750 watts (maximum configuration).

Circuitry—99% monolithic integrated circuits.

Hard Copy—up to four Teletype Model 33 or 35 Read/Only printers—or comparable equipment—may be simultaneously driven by one Display Controller when Page Print Controller options are installed.

OFFICES

ATLANTA, GEORGIA
BOSTON, MASSACHUSETTS
CHARLOTTE, NORTH CAROLINA
CHICAGO, ILLINOIS •
CINCINNATI, OHIO
CLEVELAND, OHIO •
COLUMBUS, OHIO
DALLAS, TEXAS •
DENVER, COLORADO
DES MOINES, IOWA
DETROIT, MICHIGAN
HARTFORD, CONNECTICUT
HONOLULU, HAWAII
HOUSTON, TEXAS
HUNTSVILLE, ALABAMA
INDIANAPOLIS, INDIANA
JACKSONVILLE, FLORIDA
KANSAS CITY, MISSOURI
LOS ANGELES, CALIFORNIA
LOUISVILLE, KENTUCKY
MEMPHIS, TENNESSEE
MILWAUKEE, WISCONSIN
MINNEAPOLIS, MINNESOTA
MOUNTAINSIDE, NEW JERSEY
NEW ORLEANS, LOUISIANA
NEW YORK, NEW YORK •
OKLAHOMA CITY, OKLAHOMA
OMAHA, NEBRASKA
ORLANDO, FLORIDA
PHILADELPHIA, PENNSYLVANIA
PHOENIX, ARIZONA •
PITTSBURGH, PENNSYLVANIA
PROVIDENCE, RHODE ISLAND
SACRAMENTO, CALIFORNIA
SAN FRANCISCO, CALIFORNIA •
SCHENECTADY, NEW YORK •
SEATTLE, WASHINGTON
ST. LOUIS, MISSOURI
SYRACUSE, NEW YORK
WASHINGTON, D.C. AREA •

Africa:
Bull-General Electric and Affiliates
Abidjan, Algiers, Casablanca,
Dakar, Tananarive

Australia:
Australian General Electric Pty., Ltd.
Melbourne, •Sydney •

Canada:
Canadian General Electric Co., Ltd.
Montreal, Toronto

Europe:
Bull-General Electric and Affiliates
Amsterdam, Athens, Basel,
Belgrade, Bern, Brussels,
Cologne, Copenhagen, Geneva,
Helsinki, Lisbon, London, Madrid,
Oslo, Paris, Stockholm, Vienna

Olivetti-General Electric
Bologna, •Milan, •Rome, Turin

Orient:
Bull-General Electric and Affiliates
Beirut, Istanbul, Tokyo

South America:
Bull-General Electric and Affiliates
Buenos Aires, Mexico, D.F.,
Montevideo, São Paulo

or write Drawer 270,
Phoenix, Arizona 85001

• Information Processing Centers
in these cities offer complete
computer services.

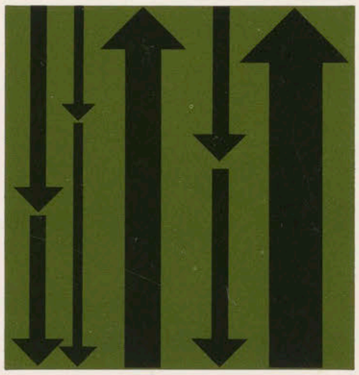
INFORMATION SYSTEMS DIVISION

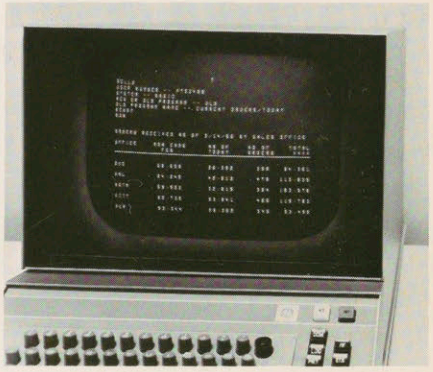
GENERAL ELECTRIC

In the construction of the equipment described General Electric Company reserves the right to modify the design for reasons of improved performance and operational flexibility.

CPB-462 (15M 5-66) LITHO IN U.S.A.

The General Electric DATANET-760 Keyboard/Display





GENERAL ELECTRIC

A New Man-Computer Relationship

Time-sharing, direct access, real time—these are the buzz-words in data processing today. These techniques are exciting and they are revolutionary—because they let the user talk directly to the computer.

Implicit in this new relationship is the requirement for a device—called a "terminal"—through which man can converse with the computer using symbols familiar to him.

The General Electric DATANET-760* is a Keyboard/Display terminal system which allows the user to talk to the computer in plain English from a distance of a few feet or thousands of miles. With the DATANET-760 at your fingertips you can solve problems, call-out and update computer stored data and, in effect, maintain an efficient and profitable control of every facet of a large and diversified organization.

Virtually anyone can operate the DATANET-760 Keyboard/Display terminal. No previous computer experience is required. General Electric time-sharing computers can talk to you in several easy-to-learn, easy-to-use languages like BASIC (developed at Dartmouth College) and FORTRAN.

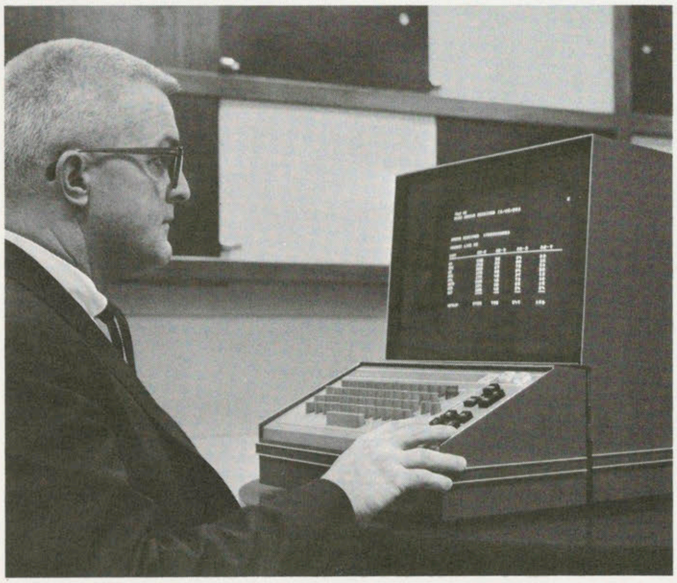
**DATANET, Registered Trademark of the General Electric Company*

Scientific / Engineering

Management Information

Inventory Control

Time-Sharing



Order Inquiry and Processing

Customer Service

Applications

Management

The DATANET-760 provides instantaneous access to computer manipulated information to allow timely, effective management decisions and more profitable operational control. A multitude of complex data—such as project status, sales analysis, customer profile and current profit and loss statements—is available in an instant. *Proved techniques enable selective information to be kept secure from those without the need-to-know.*

Engineering/Scientific

Users in industrial, educational and government research labs no longer need wait for card punching, verifying and human handling delays for solutions to computer-solved problems. The DATANET-760 allows conversation *Directly* with a time-shared computer for real-time problem solving. Problems can be composed on the display screen in the off-line mode—which allows editing and correction before transmission—and then sent to the computer in a single burst. "Scratch pad" calculations can be worked on the screen directly below the primary problem.

Finance

Not only does the DATANET-760 provide direct access to financial and statistical data, it also is a medium through which that information can be kept current. Computer stored files can be updated remotely, at the information source, as new data becomes available. The DATANET-760's formatting and editing capabilities make updating and verification a simple, quick-turnaround operation. Time, manpower and paperwork is conserved.

Marketing

With the DATANET-760, a marketing manager has instant access to such information as shelf inventory status, salesman's activity reports, and customer files. Sales analysis—by territory, product and salesman—can be accomplished in a moment from the marketer's desk. Significant data such as forecasts, effects of price variations on sales/net profit, etc., can be developed with computer assistance and displayed for decision conference or executive meetings via TV monitor or video projection. In addition, the DATANET-760 is perfectly suited for rapid turnaround order entry systems—allowing complete processing of an order, from stock check to remote print-out of shipping documents, to be handled in minutes.

Manufacturing

The DATANET-760 enables manufacturing management to maintain up-to-the-minute control over inventory, material procurement, manpower requirements, production scheduling and expediting. Simply by typing out a request on the keyboard of the DATANET-760 such information as complete vendor performance data, or the record of rejected items can be displayed instantaneously. Decisions can be made and action taken on the basis of current, accurate information.

Personnel

Under computer program control, a personnel manager can have a listing of employees and applicants with a needed skill through the DATANET-760 simply by typing in a description of the desired skill. Complete personnel records, including work history, absentee record, vacation and sick-leave time used/accrued, health insurance claims, etc.,—are available in an instant. Personnel can give managers immediate information about employees and applicants to allow most effective use of manpower.

Customer Service

Queries about charges, account status and often customer complaints are received in great numbers daily by utilities, banks and consumer credit operations. With the DATANET-760, complete customer records can be instantaneously retrieved as customer service personnel are conversing with the customer over the phone or over the counter. Much search time is saved and customers are spared irritating delays. In utility applications, requests for new service can be handled by a "new business" clerk and the new file input by her directly to the computer, via the DATANET-760. A "hook-up" order could be simultaneously printed out remotely at the utility service unit closest to the new customer.

Unique Capabilities of the DATANET-760

Complete Access to Every Character Position on the Display Page

Not only does the operator have complete access to every position on the display, but the DATANET-760 also allows the computer complete access as well. Changes or entries can be made on any line and in any character position without disturbing or rewriting data already displayed on the same line. This is a significant feature, and is especially important in file/form updating applications.

Exceptional Page Display Capacity

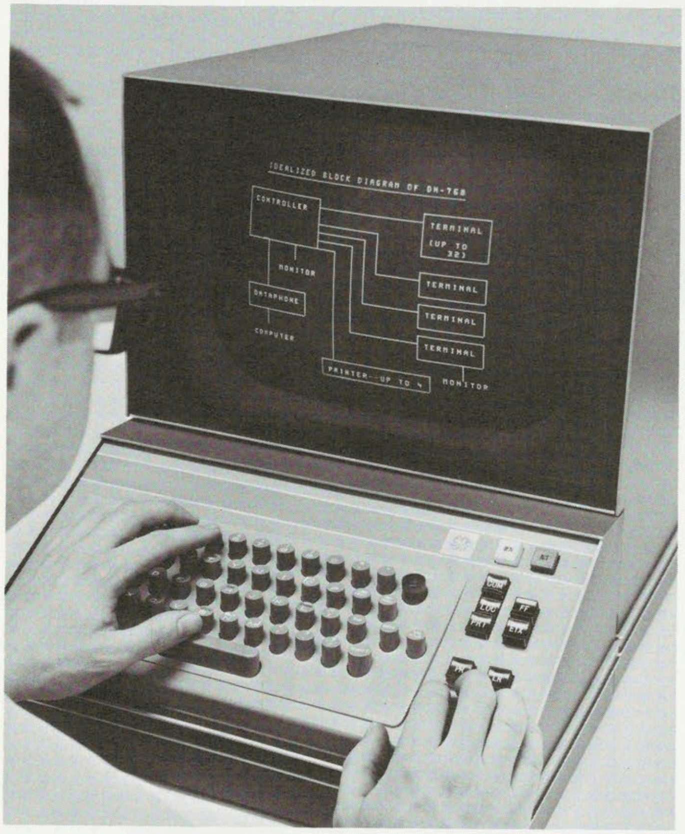
A DATANET-760 terminal may display 1196 symbols in a page array of 26 lines by 46 characters on its 14-inch TV display screen. Because of the modular design of the DATANET-760, terminals may be assigned a full display page, or segments of 4, 8 or 16 textual lines by 46 characters. This allows the most efficient utilization of the system in relation to the functional EDP responsibility assigned each terminal location.

Line Drawing Capability

Special symbols, standard on every DATANET-760 keyboard, allow the operator to draw continuous vertical and horizontal lines on the display screen, as well as segmented diagonal lines. This allows simple charts, diagrams, forms and tables to be drawn on the screen and stored in computer memory.

Standard Television Monitor Compatibility

The DATANET-760 employs a standard 525-line TV display. This standard video compatibility lets you couple many standard, industrial quality television monitors directly to the DATANET-760. This allows the sharing of information, management monitoring of input data, and simple conference hook-ups. Monitors may be co-located with the keyboard display terminal or implaced considerable distances from the terminal. *Status Monitoring—*



the standard TV compatibility also gives the user a very economical status monitoring capability. A terminal memory segment may be assigned a status monitoring function and be addressed and updated by the computer. One or more standard TV monitors may be coupled to that memory segment to provide status "monitor boards" at appropriate locations. *Video Projection*—the data display can be projected for large screen viewing with the use of standard video projection equipment.

Automatic Tab

A tab key and tab stop function let you quickly enter data in computer stored formats. The tab function has a scanning action that lets you move automatically from a position on a particular line to another tab-stop which may be located several lines below.

Error Control and Recovery

A very important feature of the DATANET-760 is its error control capability. If the computer receives a message containing an error it causes the DATANET-760 controller to automatically retransmit the message. If a predetermined number of retransmissions does not result in correction of the error, the computer automatically sends a message notifying the operator that manual recovery is necessary. If an error is detected by the DATANET-760 in a computer generated message, a blinking symbol is displayed on the display screen. This signals the operator that the message is in error and that an automatic retransmission request has been made by the DATANET-760.

Flashing Message

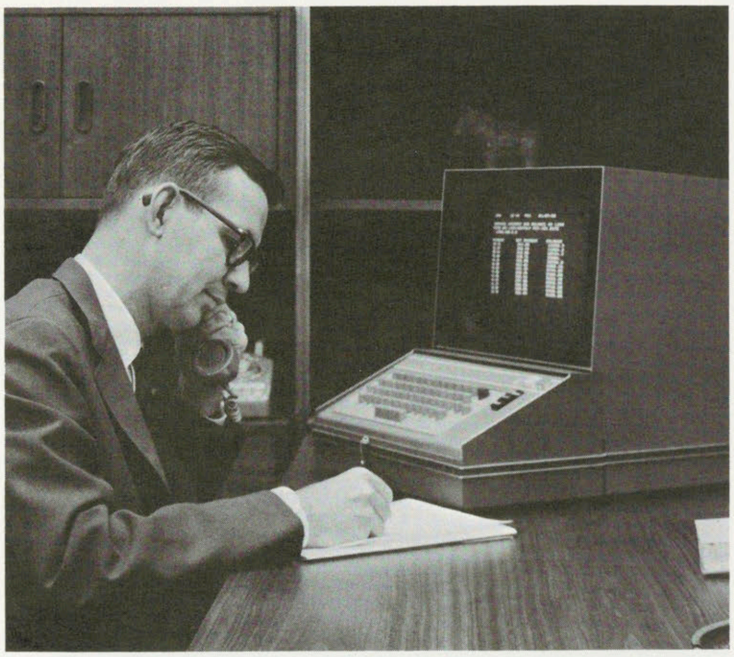
Any single character or field of characters can be displayed in a flashing, or blinking, mode to call special attention to certain information to the operator or viewer. This feature, under program control, is very useful in status reporting or, in file maintenance, to indicate the field of information to be updated.

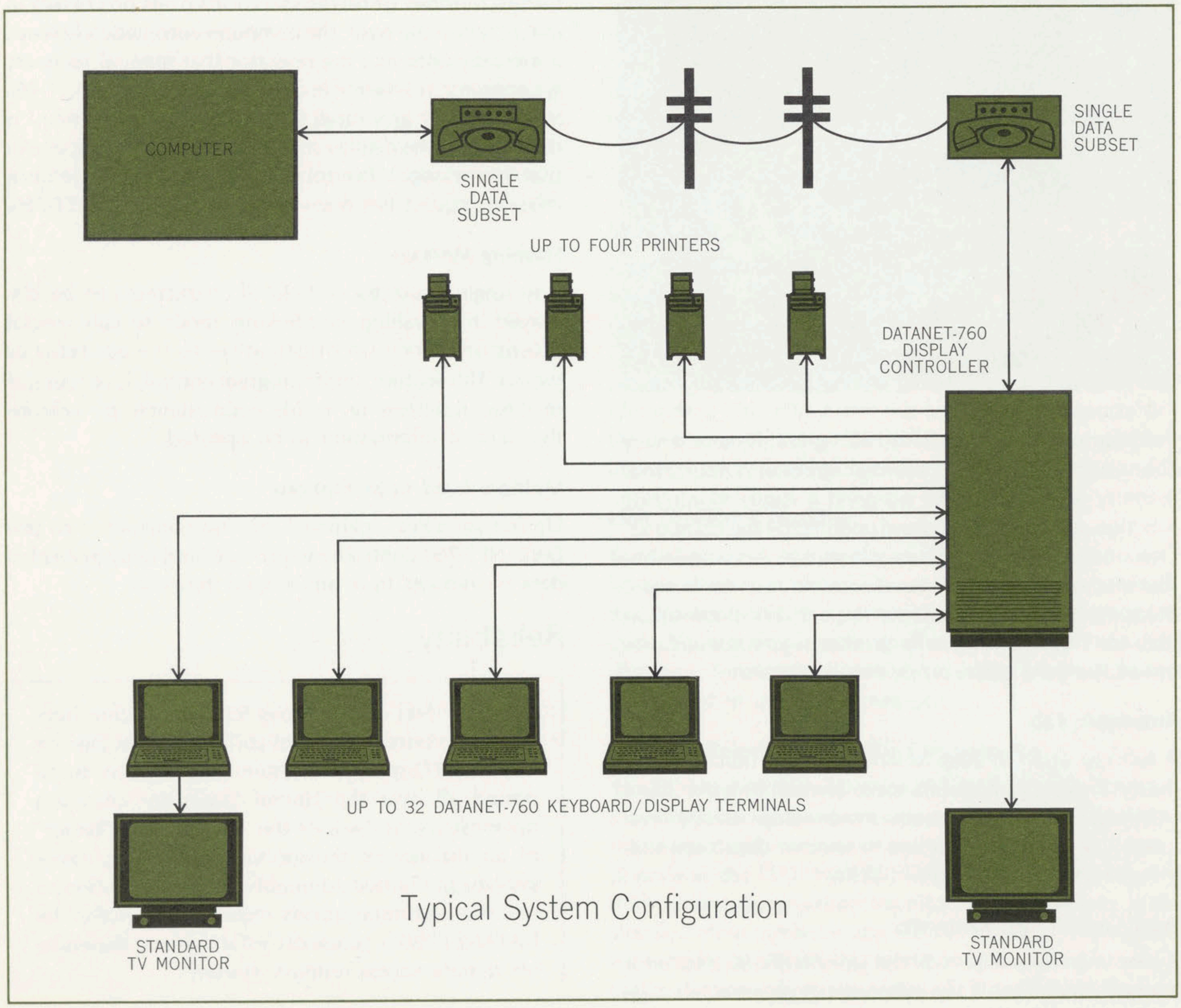
Multiple Hard Copy Options

Up to four page—printers may be connected to the DATANET-760 Controller to provide hard copy of display data on request from any display terminal.

Reliability

The DATANET-760 employs 99% monolithic integrated circuitry for high reliability. General Electric Keyboard/Display sub-systems have been transported all over the United States for countless demonstrations. Despite the "bumps" and "bangs" of all manner of transportation handling, these systems performed admirably. Special attention to realistic communications requirements make the DATANET-760 a genuinely effective and dependable remote access terminal system.





Typical System Configuration

Privacy of Data

In a time-sharing, multiprogramming computer system, it is absolutely essential that there be proven, reliable methods to prevent the accessing of confidential information by uncleared or unqualified individuals. General Electric time-sharing computers have proved techniques that do just that...with just a simple typed statement.

In addition to complete privacy, you can make a file accessible to specific persons and control the condition of their access. With a simple statement, you can make the file available only for reading, only for execution or for full access (read, write or execute).

A Flexible System

The DATANET-760 design is based on highly flexible modular concepts, allowing an information Subsystem to be tailored to individual requirements. Furthermore, a user is not forever committed to an initial configuration. Terminals may be added, their character display capacity increased or decreased, and they may be located and relocated up to 1,000 feet from the central Controller. Post-installation configuration changes or addition of modular options are simple, inexpensive in-field operations.

A typical configuration might consist of up to 32 DATANET-760 Keyboard/Display remote terminals communicating simultaneously through a DATANET-760 Display Controller with a distant computer—via a *single* telephone data communications subset. Additionally, standard TV monitors can be connected to either the Keyboard/Display terminal or the Display Controller.

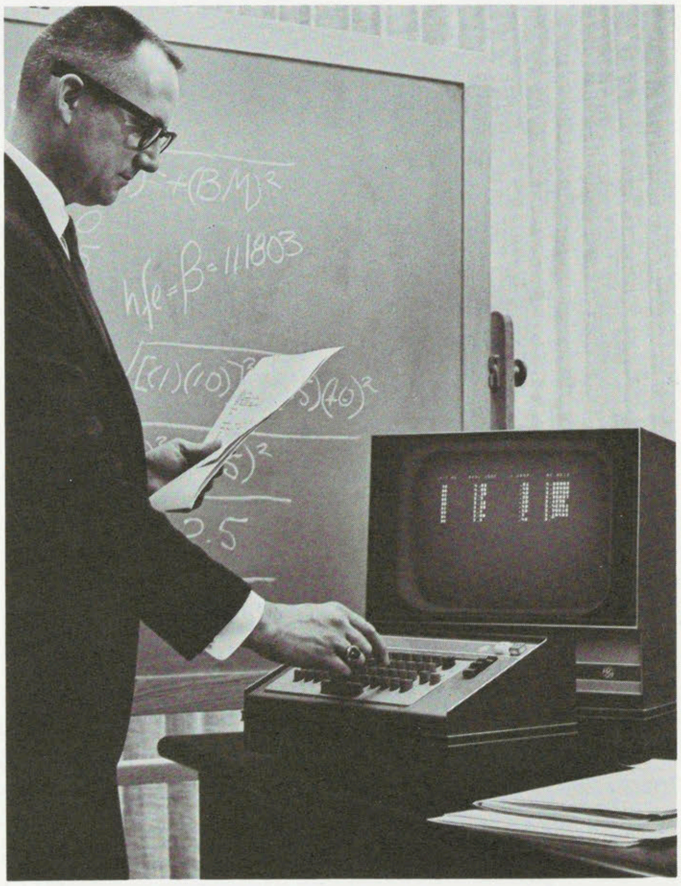
Up to four printers may be connected to the Display Controller to provide simultaneous printout of data as required from any of the Keyboard/Display terminals.

Description of Subsystem

Keyboard/Display Terminal

The DATANET-760 remote terminal consists of a television display module and a keyboard module. The keyboard and display modules may be co-located or separated by several feet. The display module contains a 14-inch cathode ray tube and supporting electronics. Brightness, contrast and other video display controls are easily accessible by the operator. The highly-reliable keyboard has a standard typewriter layout of alphanumeric symbols plus special control keys for drawing lines, transmitting data, etc.

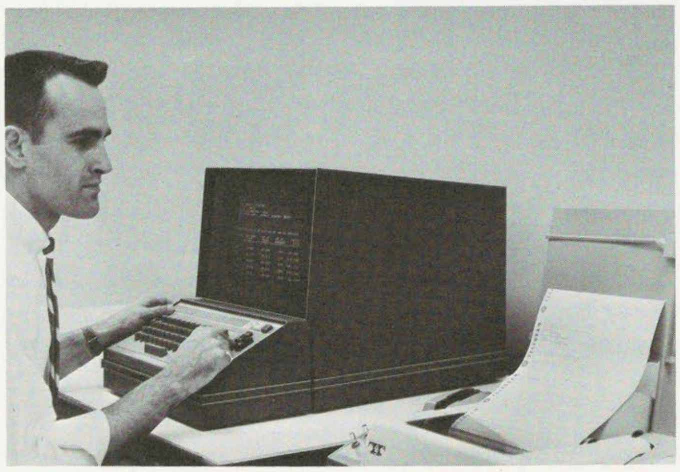
Because of the modular design of the DATANET-760, terminals may be assigned a full display page of 26 textual lines, or segments of 4, 8 or 16 lines by 46 characters.



Display Controller

The Display Controller services up to 32 DATANET-760 remote display terminals, allowing them to communicate simultaneously with the computer by direct connection or through a *single* telephone data communications subset.

The Controller cabinet contains the Basic Display Controller, buffer memories for the Keyboard/Display terminals, and optional Data Line Controller and Page Print Controller modules. Up to four buffer memories—called Terminal Memory Units—may be installed in the Display Controller cabinet. Each Terminal Memory Unit may service up to eight simultaneous access Keyboard/Display terminals. The design is modular, allowing optional modules to be added simply by "plugging them in". Keyboard/Display terminals may be located up to 1000 feet from the Display Controller.



Printer

Any printer comparable to a Teletype Model 33 or 35 Read-Only unit may be interfaced directly to the DATANET-760 or connected remotely through telephone data communications subsets. As many as four printers may be connected to one Display Controller simultaneously, allowing printout of display data from any of the Keyboard/Display terminals.

Specifications Summary

Presentation—page array of symbols (26 textual lines by 46 characters=1196 characters).

Character Repertoire—total of 64 symbols: including 26 alphabetics, 10 numerics and 28 special symbols.

Data Communications Rates—1200 or 2400 bits per second.

Self-Contained Processing and Storage—for off-line composition, editing and correction.

TV Type Display—allows use of standard TV monitors, and TV video projection and distribution equipment. High brightness in office light ambients.

Viewing Surface—14″ rectangular cathode ray tube. Format is approximately 7 x 9.3 inches (on 14″ CRT).

Symbol Matrix—7 x 10 TV lines.

Keyboard Input—standard typewriter key arrangement. Operator has complete access to every character position on the display page, as does the computer.

Character Code—**ASCII** (American Standard Code for Information Interchange).

Transmission Control—allows selection of any portion of display page for transmission to the computer.

Dimensions—Keyboard/Display Terminal: height, 17″; width, 16″; depth, 27″.
Display Controller: height, 62″; width, 27 1/4″; depth, 27 1/4″.

Power Consumption—Keyboard/Display Terminal—200 watts.
Display Controller—750 watts (maximum configuration).

Circuitry—99% monolithic integrated circuits.

Hard Copy—up to four Teletype Model 33 or 35 Read/Only printers—or comparable equipment—may be simultaneously driven by one Display Controller when Page Print Controller options are installed.

OFFICES

ATLANTA, GEORGIA
BOSTON, MASSACHUSETTS
CHARLOTTE, NORTH CAROLINA
CHICAGO, ILLINOIS •
CINCINNATI, OHIO
CLEVELAND, OHIO •
COLUMBUS, OHIO
DALLAS, TEXAS •
DENVER, COLORADO
DES MOINES, IOWA
DETROIT, MICHIGAN
HARTFORD, CONNECTICUT
HONOLULU, HAWAII
HOUSTON, TEXAS
HUNTSVILLE, ALABAMA
INDIANAPOLIS, INDIANA
JACKSONVILLE, FLORIDA
KANSAS CITY, MISSOURI
LOS ANGELES, CALIFORNIA
LOUISVILLE, KENTUCKY
MEMPHIS, TENNESSEE
MILWAUKEE, WISCONSIN
MINNEAPOLIS, MINNESOTA
MOUNTAINSIDE, NEW JERSEY
NEW ORLEANS, LOUISIANA
NEW YORK, NEW YORK •
OKLAHOMA CITY, OKLAHOMA
OMAHA, NEBRASKA
ORLANDO, FLORIDA
PHILADELPHIA, PENNSYLVANIA
PHOENIX, ARIZONA •
PITTSBURGH, PENNSYLVANIA
PROVIDENCE, RHODE ISLAND
SACRAMENTO, CALIFORNIA
SAN FRANCISCO, CALIFORNIA •
SCHENECTADY, NEW YORK •
SEATTLE, WASHINGTON
ST. LOUIS, MISSOURI
SYRACUSE, NEW YORK
WASHINGTON, D.C. AREA •

Africa:
Bull-General Electric and Affiliates
Abidjan, Algiers, Casablanca,
Dakar, Tananarive

Australia:
Australian General Electric Pty., Ltd.
Melbourne, •Sydney •

Canada:
Canadian General Electric Co., Ltd.
Montreal, Toronto

Europe:
Bull-General Electric and Affiliates
Amsterdam, Athens, Basel,
Belgrade, Bern, Brussels,
Cologne, Copenhagen, Geneva,
Helsinki, Lisbon, London, Madrid,
Oslo, Paris, Stockholm, Vienna
Olivetti-General Electric
Bologna, •Milan, •Rome, Turin

Orient:
Bull-General Electric and Affiliates
Beirut, Istanbul, Tokyo

South America:
Bull-General Electric and Affiliates
Buenos Aires, Mexico, D.F.,
Montevideo, São Paulo

or write Drawer 270,
Phoenix, Arizona 85001

•Information Processing Centers in these cities offer complete computer services.

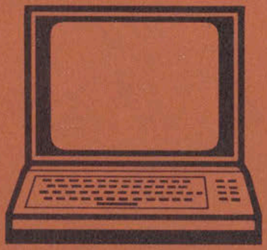
INFORMATION SYSTEMS DIVISION

GENERAL ELECTRIC

In the construction of the equipment described General Electric Company reserves the right to modify the design for reasons of improved performance and operational flexibility.

CPB-462 (15M 5-66) LITHO IN U.S.A.

DATANET-760
Keyboard/Display Subsystem Manual



GENERAL ELECTRIC

CPB — 1308A

DATANET-760 Keyboard/Display Subsystem Manual

April 1966

Rev. July 1966

INFORMATION SYSTEMS DIVISION

GENERAL ELECTRIC

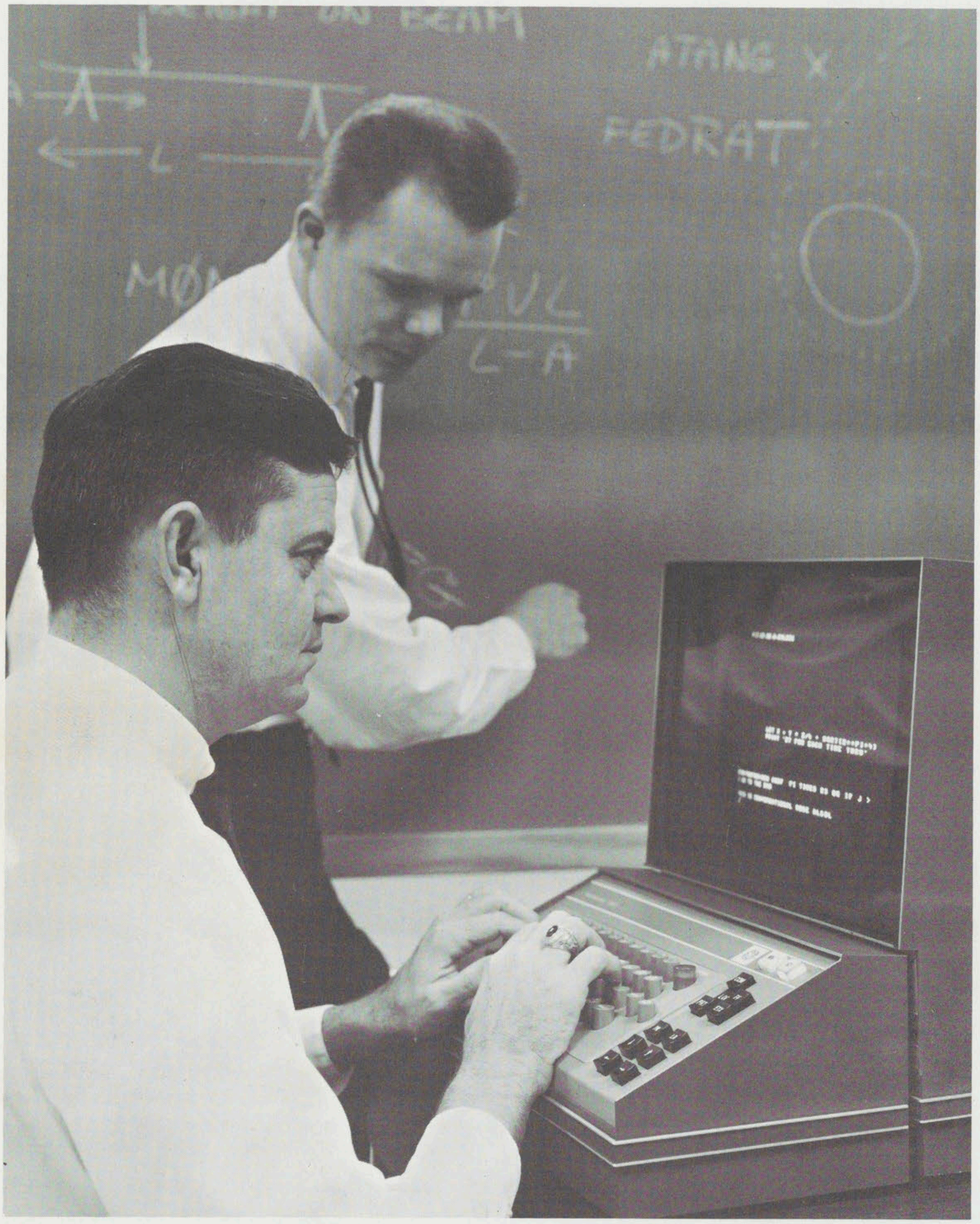
PREFACE

This Manual contains reference information on General Electric's DATANET-760 Keyboard/ Display Subsystem.

Comments on this publication may be addressed to Technical Publications, Oklahoma City Computer Operation, General Electric Company, P.O. Box 129 Oklahoma City, Oklahoma, 73101.

Contents

Illustrations



DATANET-760 Keyboard Display Unit in Use

1. SUBSYSTEM DESCRIPTION

SUBSYSTEM CONFIGURATION

The DATANET-760* Keyboard/Display is an alpha-numeric display system that provides rapid communication with computers from local or remote locations. It permits convenient entry and display of data or requests, transmission to the computer, and receipt, storage, and presentation of responses. The DATANET-760 consists of a Display Controller Unit and one or more TV-type Display Terminal Units. Up to 32 terminals, each of which may be at a remote location, may communicate with the computer through the Display Controller Unit. Up to four page printers may be connected to each Display Controller Unit (in place of Display Terminal Units) to provide simultaneous hard copy of display data from any of the terminals. A block diagram of the system is shown in Figure 1.

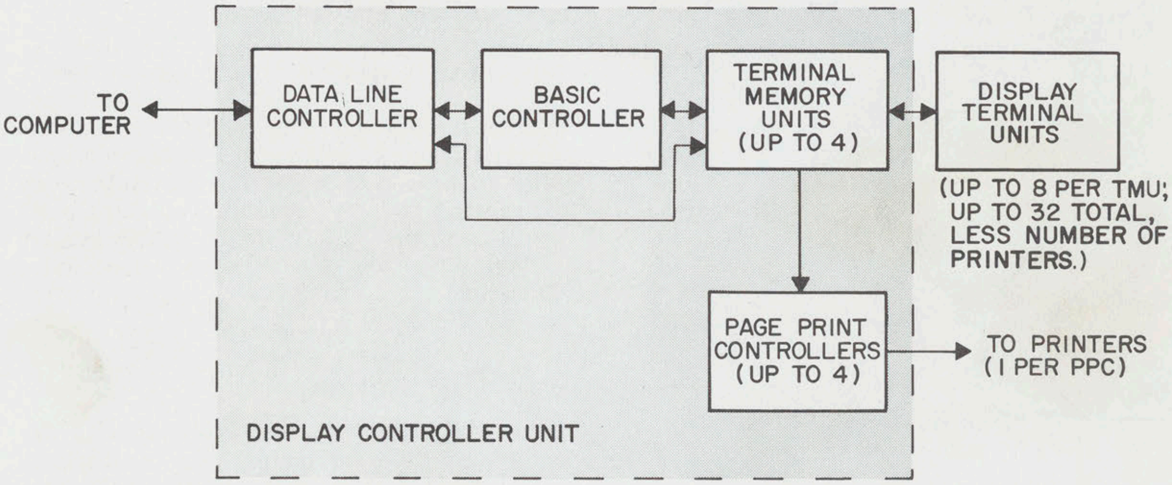

TO COMPUTER
DATA LINE CONTROLLER
BASIC CONTROLLER
TERMINAL MEMORY UNITS (UP TO 4)
DISPLAY TERMINAL UNITS
(UP TO 8 PER TMU; UP TO 32 TOTAL, LESS NUMBER OF PRINTERS.)
PAGE PRINT CONTROLLERS (UP TO 4)
TO PRINTERS (I PER PPC)
DISPLAY CONTROLLER UNIT

Figure 1. DATANET-760 Keyboard/Display, Block Diagram

The Display Controller Unit (DCU) (Figure 2) consists of a Basic Controller and up to four Terminal Memory Units, each of which can serve up to eight Display Terminal Units in simultaneous access. A Data Line Controller and multiple Page Print Controllers may be included on an optional basis.

The Display Terminal Unit (DTU) consists of a standard industrial-quality TV monitor, supporting electronics, and a typewriter-like keyboard, as shown in Figure 3. Each DTU communicates with the computer through the DCU via keyboard entries. Keyboard entries are converted to binary form and stored in the memory of the DCU. The coded characters in the memory are repetitively converted to TV video and, along with synchronizing signals, are returned to the DTU. Selected portions of the stored information are transmitted on operator command to the computer, either by direct connection or by standard digital data telephone communication service. A variety of data line communication options are available which provide a range of data rates to fit various applications.

The data display may be viewed at additional locations by coupling standard industrial-quality TV monitors to a Display Terminal Unit (DTU) or to the controller (DCU). The display can be projected for large-screen viewing by use of standard video projection equipment.

The presentation on the DTU is a fixed-format alpha-numeric display composed of up to 1196 characters and symbols stored in memory. The characters are arrayed in textual lines of 46 characters each. The number of lines in the display varies from 4 to 26 depending on the number of DTU's assigned to the Terminal Memory Unit in the DCU. The character repertoire consists of the English alphabet, Arabic numbers, punctuation marks, and special symbols, as shown in Figure 4. Four of the special symbols allow horizontal and vertical lines to be drawn for added emphasis or generating simple diagrams, charts, or tables. In addition, a flashing code allows

* DATANET is a reg. trade mark of the General Electric Company.

emergency or other important conditions to be emphasized.

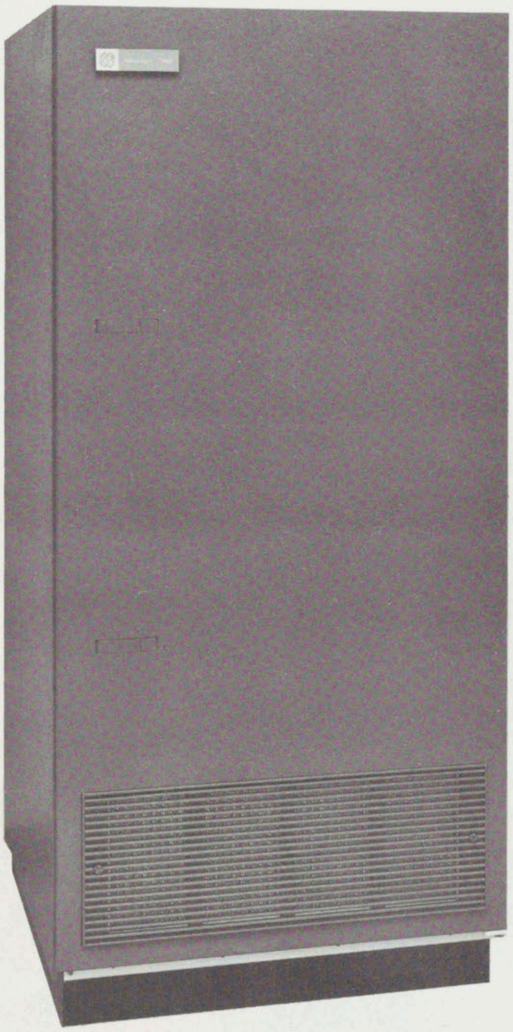


Figure 2. Display Controller Unit

OPERATION

The operator enters data by typing on the keyboard as on an office typewriter. The characters and symbols are instantaneously displayed as they are typed. A special entry marker appears on the display to indicate the location of the next character to be entered. The marker automatically indexes with each character entry or may be manually spaced forward or backward, and up or down. It may also be reset to the first character position of the page or textual line. In addition to providing repetitive character entry capability, the (repeat) REPT key allows a continuous scanning movement of the marker. Changes or corrections are made by relocating the marker to the erroneous character and typing the correct one. Erasure of the entire display is accomplished by a single control operation.

A TAB key allows the operator to quickly and efficiently enter information into an operator-composed or computer-stored format. Depressing the TAB key causes the entry marker to scan the display face until it finds a vertical line, where it stops. These vertical lines, which serve as tab-stop markers, can be positioned anywhere on the display surface by the operator or the computer.

The operator completes the composing, verifying, and correction of the entry with the system off-line. When satisfied that the information is correct, the operator locates the "end of text" (ETX) symbol opposite the last line of characters to be transmitted; then returns the entry marker to the first character to be sent; and depresses the transmit key. Successive characters are transmitted up to the "end of text" symbol.

Responses from the computer are stored in the memory of the DCU and immediately appear on the display. The operator may obtain a printed copy of any portion of his display - when the page printing option is installed - by depressing the PRT (print) pushbutton after positioning the "end of text" (ETX) symbol and entry marker as in the data transmission operation.

APPLICATION

The DATANET-760 has application in any commercial, government, or educational function requiring storage, retrieval, communication, or display or human or computer-generated data.

Some typical applications include the following:

- Data entry and handling in automatic check-out systems
- Data handling of reservations, and other transportation and lodging information
- Production and inventory control, status reporting, forecasting, and planning
- Transmission, acquisition, storage, and read-out of media news copy
- Checking of account status, deposits, and withdrawals
- Accumulation, sorting, and read-out of weather bureau data

- Insurance claim adjustment and policy file maintenance and search
- Customer account maintenance, order processing, and invoicing
- Control of personnel records, group insurance files, payroll data, manpower scheduling, and costing
- Library information retrieval, classroom instruction, and computer training

In all applications the DATANET-760 provides a direct link to the computer and under program control, allows real-time problem solving, and instantaneous information retrieval.

The DTU presentation for a representative management application is shown in Figure 5. In this application the operator would request a general form stored in the computer, enter the specific data--manpower estimates in the display shown--and transmit the data to the computer for processing. Totals calculated by the computer would be displayed, and, as the operator made changes, the new totals would appear. Any DTU in the system could call up the complete display showing current data at any time.

MODES OF OPERATION

There are two modes of operation for a DATANET-760 Display Terminal Unit (DTU) - "local" and "receive". The "local" mode is

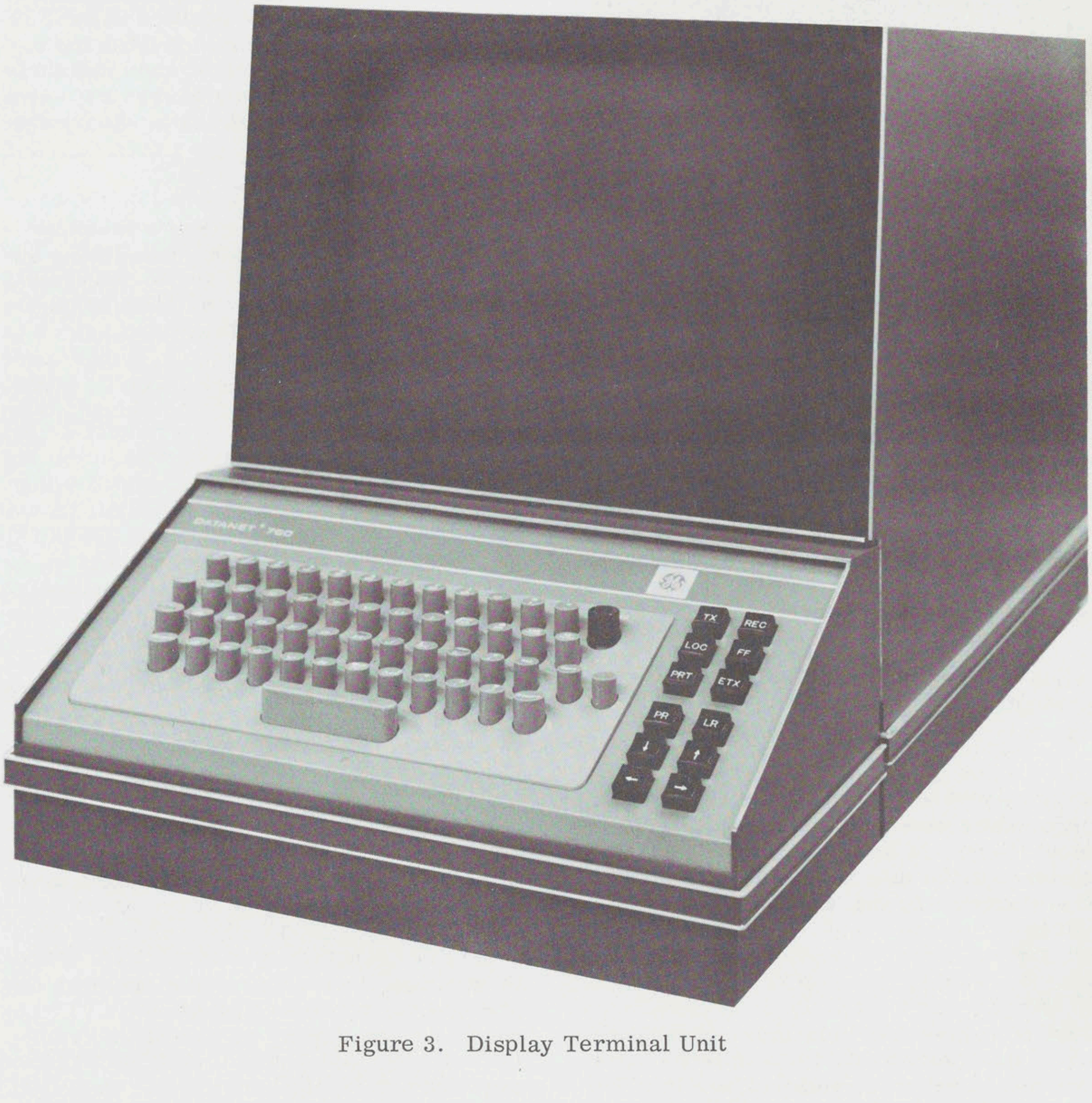


Figure 3. Display Terminal Unit

an off-line compose mode during which the operator is free to write on his display without interruption by the computer. The "receive" mode operates in the same manner as the "local" mode with the exception that the DTU is able to receive a computer message via the communication line. During either mode, the operator is able to transmit to the computer or print all or any part of the data appearing on his display. A letter is displayed in the left-hand margin of the display opposite the first line of data, as shown in Figure 6, to indicate the mode of operation or that transmission or printing is taking place.

0	1	2	3	4	5	6	7	8	9
:	;	A	B	C	D	E	F	G	H
I	J	K	L	M	N	O	P	Q	R
S	T	U	V	W	X	Y	Z	<	=
>	?	,	-	'	+	*	&	(	)
%	$	#	"	.	!	/	bk	sp	

NOTE: bk = blink following characters,
sp = space

LINE SYMBOLS (letter is shown for comparison)

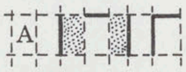


Figure 4. Character Repertoire

Local (Off-Line Compose) Mode

The DTU goes into this mode of operation when the local (LOC) pushbutton is depressed. During this mode, the DTU will not be able to receive data via the telephone communication line. The operator is free to write on the display through his keyboard without being affected by the communication line, and to transmit displayed data.

When a DTU is operating in this mode, the presentation will have an "L" in the mode symbol position.

Upon receipt of a computer-generated message addressed to a DTU operating in the local mode, a message indicating a busy status will be automatically transmitted to the computer by the Display Controller Unit (DCU).

Receive Mode

The DTU goes into this mode of operation when the receive (REC) pushbutton is depressed. The mode symbol is the letter "R". During this mode, the DTU is able to transmit or receive data via the communication line. In this mode, the operator is allowed to compose or type on the display as long as he is not transmitting or receiving a computer message. However, while composing on the display, he is subject to interruption by a computer-generated message. During transmission or reception of a message, keyboard entry is prevented, unless the message is an automatic acknowledgment (ACK).

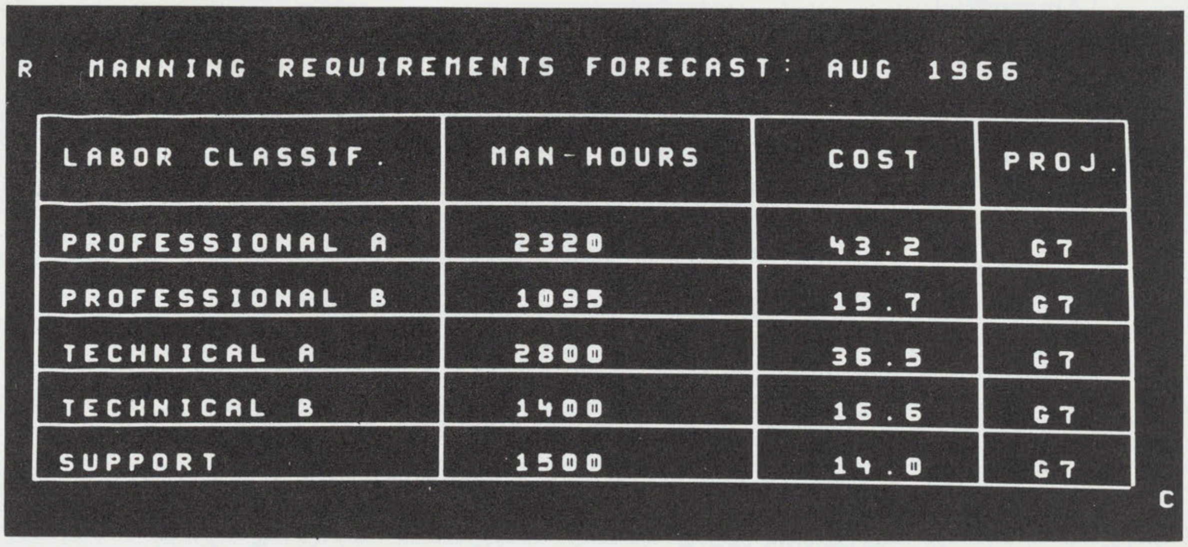
Received data begins loading at the position of the entry marker and proceeds sequentially from there with each character entry. The computer can position the entry marker to any position on the display to begin a message. To end each message the computer transmits an "end of text" (ETX) code. However, the ETX in a computer-generated message does not affect the position of the ETX symbol on the display. This symbol appears as the letter "C" in the right-hand margin of the DTU display, and is movable to a position opposite any line of text. It is always displayed.

If an error condition is detected on the communication line during the reception of a message, the mode symbol will flash. The flashing will continue until an error-free message is received. If the DTU is busy, the error condition is ignored.

DATA TRANSMISSION

When the DTU is in either the receive or the local mode, the operator is able to initiate data transmission on the communication line. The operator prepares a message for transmission by positioning the end of text (ETX) symbol and returning the entry marker to the first character of the message. Positioning the ETX symbol in preparation for transmission is accomplished by moving the entry marker to any character position of the last line of text in the message and depressing the ETX key. The ETX symbol will then move from its present position to opposite the last line of text in the message.

Once the message is prepared for transmission, the operator depresses the transmit (TX) pushbutton. Transmission begins at the character identified by the entry marker position, proceeds sequentially on the page, character by character, until the ETX symbol is reached. As soon as the operator depresses his transmit pushbutton, a "T" appears on the display as the Mode Symbol. Upon receipt of an acknowledge (ACK) message from the computer the "T" is erased, the mode symbol appears as an "R", and the



R MANNING REQUIREMENTS FORECAST: AUG 1966

LABOR CLASSIF.	MAN-HOURS	COST	PROJ.
PROFESSIONAL A	2320	43.2	G7
PROFESSIONAL B	1095	15.7	G7
TECHNICAL A	2800	36.5	G7
TECHNICAL B	1400	16.6	G7
SUPPORT	1500	14.0	G7

C

Figure 5. Typical Display

DTU is placed in the receive mode. If the operator changes the mode to local before the acknowledgment is received, the "L" mode symbol remains on the display and the DTU remains in the local mode (see note).

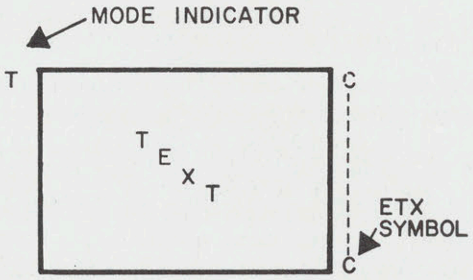


Figure 6. Display Showing Mode Symbol Position

During transmission of a message consisting of more than one line of text, the codes for carriage return and line feed are automatically transmitted at the end of each line (except the final line). Immediately after the operator depresses the transmit pushbutton, keyboard entry is prevented until message transmission is complete. After transmission is complete, the entry marker appears in the first character position of the line following the ETX symbol.

PRINTING

When the DTU is in either the receive or the local mode, the operator may request a printed copy of any portion of his display. Preparation of a message to be printed is the same operationally as for transmission. The operator prepares a message by positioning the ETX symbol opposite the last line of text to be printed and returning the entry marker to the first character of the message.

Once the message is prepared, the operator depresses the print (PRT) pushbutton. This action prevents further keyboard entry until message transmission is complete. The letter "P" appears as the mode symbol indicating to the operator that the request for hard copy has been made. Upon receipt of the acknowledge message from the computer the "P" is erased, and the DTU is free to perform other functions. When the "P" is erased, the mode symbol appears as an "R", and the DTU is placed in the receive mode. If the operator changes the mode to local before the acknowledgment is received, the "L" mode symbol remains on the display and the DTU remains in the local mode (see note).

Note

It is not recommended that the system be defeated in this manner, since NAK responses will be ignored. By waiting for the ACK (T change to R) the operator has positive verification that the computer received the message correctly.

2. DISPLAY TERMINAL UNIT

The Display Terminal Unit (DTU) consists of two basic elements, the keyboard and the CRT display, as shown in Figure 7.

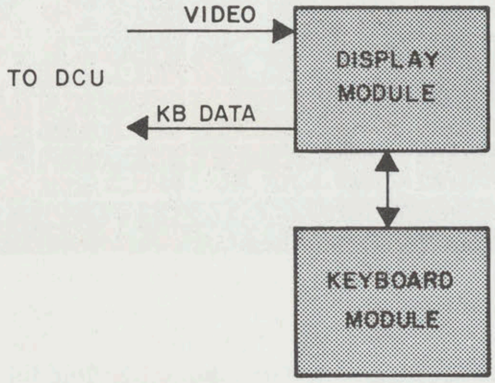


Figure 7. DTU Display Module and Keyboard Module

KEYBOARD MODULE

The keyboard module provides the standard Alpha-Numeric Key group, the Command Key group, and the power switch. An optional group, the Entry Marker group, may also be included. The keyboard layout, including the optional key group, is shown in Figure 8.

The keyboard module is a physically separate unit from the display module. The standard cable length provided between the keyboard and display modules is 4 feet; however, for special applications the cable distance may be up to 100 feet.

Alpha-Numeric Keys

The following keys cause the entry and display of the corresponding character or symbol with a single key action:

Key Label and Symbol Displayed

A	H	O	V	2	9
B	I	P	W	3	:
C	J	Q	X	4	;
D	K	R	Y	5	,
E	L	S	Z	6	-
F	M	T	0	7	/
G	N	U	1	8	.

The space bar erases the character under the entry marker, if any, displays a blank space, and moves the entry marker one space forward.

The following symbols are generated by using the SHIFT key and the designated symbol key.

Key Label and Symbol Displayed

!	&	+	
"	'	<	-
#	(	=	┌
$	)	>	╷
%	*	?	╵

The BLK (Blink) key enters the blink code in memory (displayed as a space), causing the blinking of all following symbols up to the first space or up to and including character position 46.

Entry Marker Control Keys

The entry marker control operations listed in Figure 8 are performed by using the control key and the designated operation key. In the case of the tab, line feed, and line return, it is not necessary to depress the control key.

Entry Marker Control Group (Optional)

This group of keys will provide identical operations to the Entry Marker Control Keys; however, each operation may be performed with a single pushbutton. This group of keys is physically separated from the Alpha-Numeric area of the keyboard module. Figure 8 lists this group also.

DISPLAY MODULE

The display module is intended to be used as a viewing device during composition and editing of information prior to transmission and for the presentation of response data received from the computer.

The display is based on standard television techniques. A two field interlaced scan system is used in presenting the display data. The cathode ray tube is coated with P4 phosphor, which is a high efficiency medium-short persistence phosphor with white fluorescence (5250 Å). This is the same phosphor that is used in standard black and white home television receivers.

Brightness

The display brightness (light emitted by the CRT beam trace) exceeds 75 foot-lamberts. Since the contrast ratio is a function of the direct incident screen light (from the room) each room lighting condition will affect the absolute contrast ratio measurement. However, in a room with 70 foot-candles of diffused ambient lighting the contrast ratio will be at least 20:1.

Controls

The controls provided for the display module are of two types: Operator controls (shown in Figure 9 and listed in Figure 10) which are readily accessible to and for use by the operator, and maintenance controls (listed in Figure 11) which are intended for use only by qualified maintenance personnel.

Screen Size

The display module has a 14-inch (diagonal measurement) screen cathode ray tube. Extension monitors are available on special request with 14-inch and 23-inch screens. (See page 9)

Textual Display

The display format consists of from 4 lines to 26 lines of text data, each line consisting of 46 characters. The number of lines is a function of the number of DTU's per Terminal Memory Unit. The full format size (26 lines of 46 characters) for a 14-inch screen is 6.3 by 8.0 inches.

For formats of fewer than 26 lines the format height is reduced proportionately and will be approximately centered on the screen. Measurements are nominal values.

Mode Indicator Display

A DTU mode indicator is displayed in the character space preceding the first text character position of the first format line, as shown in Figure 12.

The mode symbology and meaning are listed in Figure 13.

The End of Text symbol (C) is located in the right margin following the 46th text character position as shown in Figure 12. It may be positioned at the end of any line.

Character Size

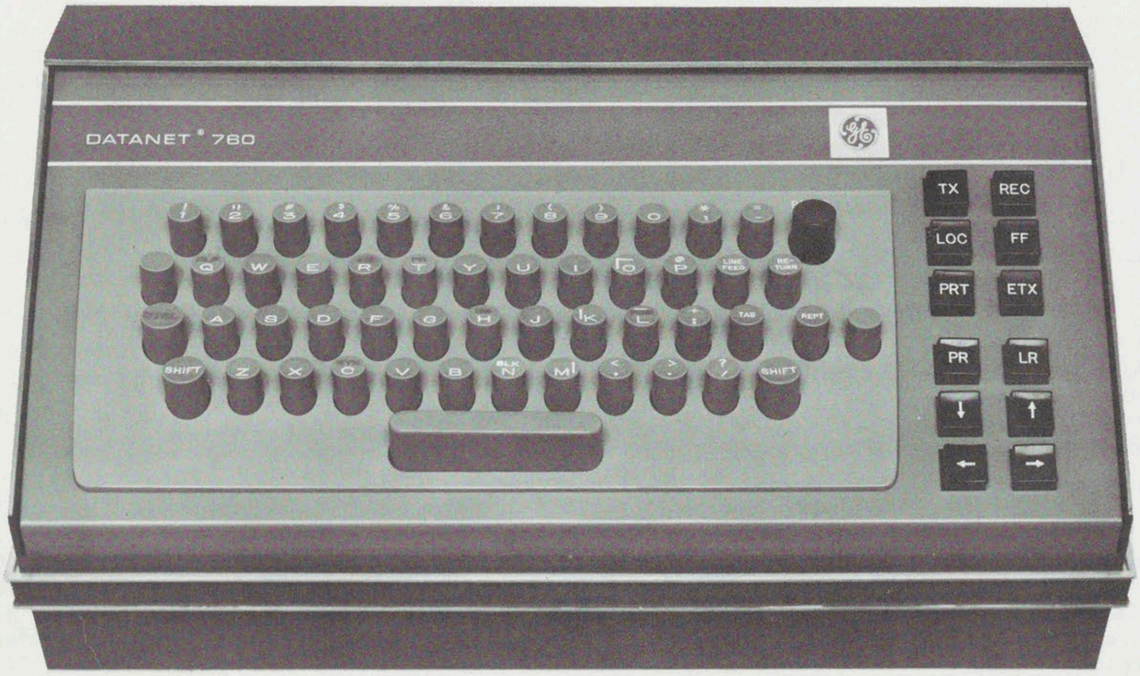
Character size is a function of the format size and hence the screen size. The following table shows the approximate character size (standard, full sized characters) for 14-inch and 23-inch screens:

Screen Size (Diagonal)	Standard Character Size (Height x Width)
14 inch	0.16 x 0.12 inch
23 inch	0.26 x 0.19 inch

The preceding values are nominal.

Monitor Unit

Monitor units may be connected to a DTU to provide additional viewing positions duplicating the information displayed on the DTU. Up to four receive-only monitors can be connected in series to the video jack on the DTU. Monitor units may also be connected to a video output of a TMU for receive only - i.e., in place of a DTU. Up to five monitors may be driven from a TMU video output. Monitor units are available on special request.



CONTROL KEYS

Key Label	Operation
BS	Backspace entry marker one space.
FS	Forward space entry marker one space.
LINE FEED	Line Feed - Moves entry marker down one line at the same character position.
ETX	End of Text - Enters ETX (C) symbol on display at end of line on which the entry marker is located.
RLF	Reverse Line Feed - Moves entry marker up one line at the same character position.
PR	Page Return - Returns entry marker to the first character position on the first line.
RETURN	Carriage Return - Moves entry marker to the first character position of the line.
FORM	Form Feed (clear memory) - Erases the entire display except for the mode character, ETX symbol, and the optional Function characters. This command also automatically page returns the entry marker.
TAB	Moves entry marker from its initial position to the character position following the next vertical line symbol.

Figure 8. Keyboard Layout and Key Label

COMMAND KEYS

Key Label	Operation
PRT	Print - Causes the information displayed to be printed. Symbol "P" appears in mode display position when print command is given, and the marker is moved to the first character position of the line following the ETX symbol when the information transfer takes place.
LOC	Local - Allows the entry of data only from the keyboard. Symbol "L" appears in mode display position when local operation is selected.
REC	Communication - Allows the computer to enter or update display at any time. The entry of data from the keyboard is also allowed. Symbol "R" is displayed in the mode display position when REC operation is selected.
TX	Transmit - Request transmission of message to the computer. When transmission is requested, keyboard entry is prevented until the transmission is completed. The symbol "T" appears in the mode display position upon the TX command, and the entry marker is positioned after the ETX symbol when the transmission is complete.

ENTRY MARKER CONTROL GROUP (OPTIONAL)

Key Label	Operation
←	Backspace entry marker one space. (Repeated if held down.)
→	Forward space entry marker one space. (Repeated if held down.)
↓	Line Feed - Moves entry marker down one line at the same character position. If the entry marker is initially located on the last line of the format, it will automatically return to the top line. (Repeated if held down.)
↑	Reverse Line Feed - Moves entry marker up one line at the same character position. When the entry marker reaches the top line of the display, the Reverse Line Feed operation has no effect. (Repeated if held down.)
PR	Page Return - Returns entry marker to the first character position on the first line.
LR	Line Return - Moves entry marker to the first character position of the line.
ETX	End of Text - Enters ETX symbol on display at end of line on which the entry marker appears.
FF	Form Feed (clear memory) - Erases the entire display except the mode character, ETX symbol, and the optional Function characters. This command automatically page returns the entry marker.

Figure 8. Keyboard Layout and Key Label (Cont)

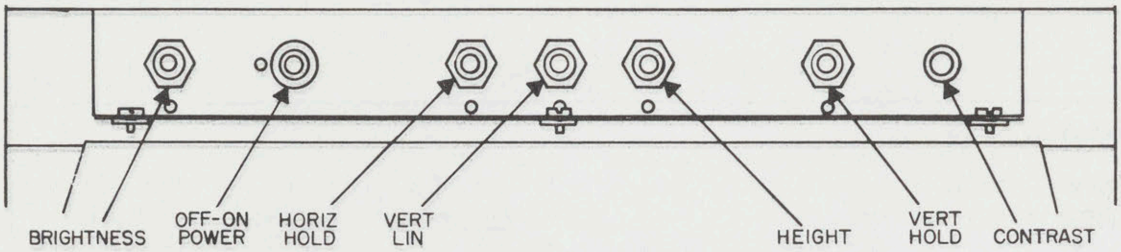


Figure 9. Operator Controls (Beneath Screen)

BRIGHTNESS - Controls background display brightness.

OFF-ON POWER - Controls power to both the display module and the keyboard.

HORIZONTAL HOLD - Controls the horizontal synchronization of the display.

VERTICAL LINEARITY - Controls relative height of first and last display lines.

HEIGHT - Controls height of total display.

VERTICAL HOLD - Controls the vertical synchronization of the display.

CONTRAST - Controls character display brightness.

Figure 10. Operator Control Listing

FOCUS

WIDTH

HORIZONTAL LINEARITY

HORIZONTAL DRIVE

VERTICAL FEEDBACK

VIDEO AMPLIFIER PEAKING ADJUSTMENTS

HIGH VOLTAGE ADJUSTMENTS

LOW VOLTAGE ADJUSTMENTS

PICTURE CENTERING

Figure 11. Maintenance Controls

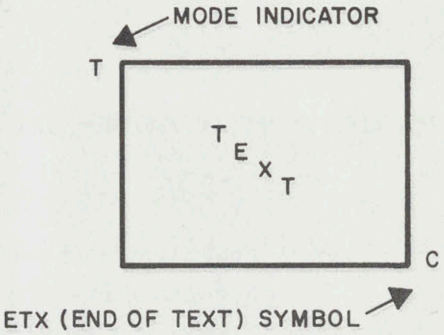


Figure 12. Mode Indicator and End of Text Symbol Positions

Mode Indicator		Meaning
T	-	Indicates that the DTU operator has depressed the Transmit (TX) pushbutton and that the computer generated acknowledgment (ACK) message has not yet been received by the DATANET-760.
Blinking Mode Indicator	-	Indicates that an error was recorded during the last received message from the computer to the DTU.
P	-	Indicates that the operator has depressed the print (PRT) pushbutton and that the computer-generated "ACK" message has not yet been received by the DATANET-760.
L	-	Indicates that the operator has pressed the local (LOC) pushbutton, which places his DTU in the off-line compose mode.
R	-	Indicates that the operator has pressed the receiver (REC) pushbutton, which places his DTU in the receive mode.

Figure 13. Mode Indicators and Meanings

3. DISPLAY CONTROLLER

The Display Controller is a centrally located equipment capable of controlling up to 32 remotely located terminals.

The Display Controller is composed of one Display Controller Unit (DCU) which consists of the Controller Cabinet and the Basic Controller; one Data Line Controller (DLC); a minimum of one and a maximum of four Terminal Memory Units (TMU's); and up to four Page Print Controllers (PPC's) on an optional basis. (See Figure 14). The minimum operable configuration is one DCU, one DLC, and one TMU; the maximum is one DCU, one DLC, four TMU's, and four PPC's.

BASIC CONTROLLER

The Basic Controller (part of the DCU) includes the following functions:

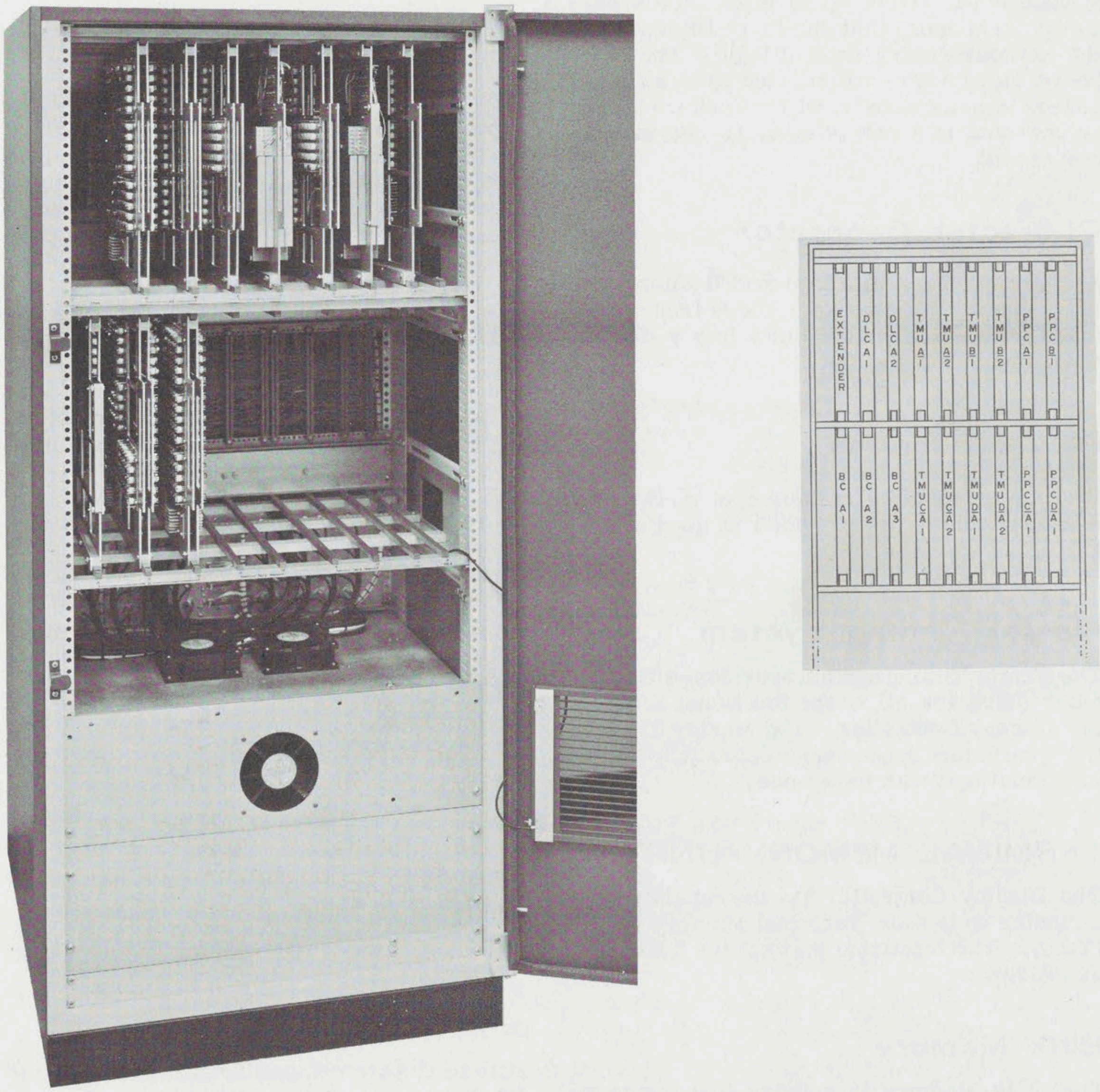


Figure 14. Rear View of Display Controller Unit

Keyboard Entry Control

The Keyboard Entry Control logic is a data multiplexer which accepts inputs from the DTU Keyboards. Each input from the keyboard is decoded to determine if it is a command code or a data character to be stored. Command characters are used to control the position of the entry marker, record in memory the DTU mode, or clear the entire display. Data characters are routed to a location in memory corresponding to the character position under the entry marker on the display. After each character entry, the entry marker automatically indexes to the next consecutive position on the display.

Keyboard data from up to eight DTU's per Terminal Memory Unit can be multiplexed by the keyboard entry control logic. The keyboard entry logic will service up to 32 keyboards simultaneously, where each keyboard is operated at a rate of up to 15 characters per second.

Character Generator

A Character Generator is provided which converts the six-bit character codes from up to four Terminal Memory Units into a digital video signal for display.

Power Supplies

DC power supplies sufficient to power the functional units are provided as part of the DCU.

Master Timing System

The Master Timing System provides sufficient clock drive for all of the functional units of the Display Controller. The Master Timing Unit guarantees synchronous operation of each functional unit with the system.

TERMINAL MEMORY UNIT

The Display Controller has the capability of including up to four Terminal Memory Units (TMU). The functional units of the TMU are as follows:

Bulk Memory

The Bulk Memory is a delay line memory which is capable of storing up to 1472 six bit character codes. The memory data can be partitioned in segments of character data so that each segment is displayed on a different DTU. The following partitioning configurations are provided:

- Eight segments of data which allow the display of four 46 character lines on each of up to eight DTU's.
- Four segments of data which allow the display of eight 46 character lines on each of up to four DTU's
- Two segments of data which allow the display of sixteen 46 character lines on each of up to two DTU's
- One segment of data which allows the display of twenty-six 46 character lines on one DTU

Video Distribution

The Video Distribution logic of the TMU accepts data from the character generator of the Basic Controller and forms these inputs into composite video signals to drive up to eight DTU's. There is a composite video signal corresponding to each memory partitioned segment described above.

The Video Distribution logic provides the video blanking required to prevent a given DTU's data from appearing on any other DTU, and causes the presentation of each DTU to be approximately centered in the middle of the display screen. It also provides for the display of an entry marker symbol on each DTU screen. The entry marker always appears over the location of the next character to be entered. The entry marker takes the form of a horizontal line whose length is approximately the width of a character position. To make the entry marker easier to locate on the display it flashes at a 3 to 5 cycle per second rate.

DATA LINE CONTROLLER

The Data Line Controller (DLC) provides the facility to interface the Basic Controller to a remote station or computer. There are two types of DLC's:

1) 1200 Bit/Sec - Half Duplex Asynchronous

2) 2000/2400 Bit/Sec - Half Duplex Synchronous

Both DLC's are compatible with EIA Standard RS-232A. Interface connections are made through a single connector. A Longitudinal Parity Check option is available.

1200 Bit/Sec Option

The 1200 Bit/Sec half duplex asynchronous option provides two-way, non-simultaneous transfer of information to and from the DATANET-760 at a bit rate of 1200 bit/sec. The interface is compatible with Data-Phone Models 202C and 202D or equivalent.

2000/2400 Bit/Sec Option

The 2000/2400 bit/sec half duplex synchronous option provides two-way, non-simultaneous, synchronous transfer of information to and from the DATANET-760 at a bit rate of 2000 or 2400 bit/sec. The bit rate is selectable at the time of installation by choice of the digital data set. Transmit and receive timing at the data bit rate is provided by the digital data set when present or by the computer station when direct drive option is used. The interface is compatible with Data-Phone Model 201A and 201B or equivalent.

Automatic Message Initiation

Messages may be automatically transmitted from a terminal without operator action as a result of certain conditions in a received message for that terminal. The conditions which will cause automatic initiation of a message from a terminal are as follows:

1) Completion of a received message containing a Text Message (NUL) status character. An acknowledgment message is transmitted.

2) Completion of a received message containing a detected error. A negative acknowledgment message is transmitted.

3) Completion of a received message containing a Negative Acknowledge (NAK) status character. A retransmission of the previously transmitted message is initiated. For compatibility with similar DATANET equipment, an Enquiry (ENQ) character must be included anywhere in the text of the "NAK" message.

4) Completion of a received message for a terminal which is busy. A terminal is busy if any of the following conditions are present:

   a) The terminal is in the local mode.

   b) A transmit or print request from a DTU keyboard has not yet been serviced.

   c) An acknowledgment or negative acknowledgment is awaiting transmission.

   d) A retransmission request has not yet been processed.

   e) If the terminal is a page printer and a previous message addressed to the Page Printer Controller is still being printed.

   f) A DTU is not busy due to these conditions if the message is an "ACK" response.

Special handling of messages addressed to the Page Printer Controller is recommended to achieve maximum usage of the printer. Messages addressed to the PPC will not be acknowledged until printing is complete. Thus, a "BSY" response should not be used to test the PPC status.

Transmission Frame Half Duplex

START UP AND QUIESCENT TRANSMISSION FRAMES. At initial turn on, assuming no information transfer has been requested by an operator, the DLC will time out the absence of a transmission frame from the remote station. When the timer runs out, the DLC will initiate a transmission frame on the communication line. The timer is set for 4.226 +0, - 0.266 seconds, exclusive of phone line turnaround time. The transmission frame is composed of two characters, Start of Header (SOH) followed by End of Transmission (EOT).

The completion of the received transmission frame at either the DLC or the computer immediately causes the receiving party to initiate a transmission frame in reply. When either end has no information to transfer, a simple "SOH" followed by a "EOT" constitutes the transmission frame.

INFORMATION TRANSMISSION FRAMES ORIGINATED FROM DLC. Prior to transmission, and at the end of each individual message, a scan is made of all DTU's in descending order of DTU address (DTU-8 to DTU-1 and TMU-D to TMU-A) until one has been found with a request for message transmission. The scanner will remain locked to that DTU until all transmit conditions have been cleared.

The transmission frame will begin with the transmission of the selected DTU's message.

After the message has been transmitted, the scanner will be stepped once through all remaining DTU's with smaller addresses in that TMU. All of the remaining DTU's with smaller addresses in that TMU which have messages to be transmitted will have their messages included in the transmission frame.

After all DTU's with smaller addresses in the selected TMU have been scanned and/or serviced, the transmission frame will be terminated by the generation of an EOT.

In half-duplex 1200-bps DLC's (DLC-760), all remaining DTU's of a TMU can be scanned without injecting time fill between messages. In half-duplex 2400-bps DLC's (DLC-765), a single time-fill character may follow the longitudinal-parity character prior to the next character should the memory segment of a following DTU having a transmission be widely separated from the prior segment.

INFORMATION TRANSMISSION FRAMES ORIGINATED FROM COMPUTER. An information transmission frame originating from the computer may contain messages for DTU's in any or all TMU's. The order and number of messages in a transmission frame originating from the computer may arbitrarily be determined by the computer program.

Transmission frames are originated upon the termination of a received transmission frame from the DLC. (Note: See appendices "C" through "G").

4. PAGE PRINT CONTROLLER

The Page Print Controller (PPC) is an optional module which is designed to drive one Model 33 or 35 Send-Receive or Receive-Only Teletypewriter Page Printer, or comparable equipment.

Each PPC module receives data inputs from a TMU. The PPC utilizes TMU memory space normally allocated to a DTU. Therefore, for each PPC module included in the Display Controller, the maximum number of DTU's allowed is reduced by one. The PPC is designed so that it can be wired to accept data from any TMU and utilize memory space corresponding to any DTU position.

The memory space utilized by a PPC can be addressed and loaded by the computer in the same manner that a DTU memory space is loaded. Print requests from a DTU are transmitted to the computer via the DLC as a normal transmitted message. The computer receives and buffers each print request message until it is able to retransmit it to the desired PPC memory section to obtain a hard copy.

Since the computer buffers the print request messages, the computer program can be designed to generate print message labeling or add other types of data to a print message to closely fit the needs of each application. This allows the operator to proceed with additional work.

The output interface of the PPC is the EIA standard RS232A. Interface signals and output character coding are given in the Appendix. Each character is bracketed by one start bit and two stop bits. The output data transfer rate is 110 bits per second.

MEMORY SIZE

Since the PPC is added in place of a DTU and utilizes a DTU memory section, its memory capacity is equal to that of the DTU position it occupies. Even though a single print message is limited to the number of text rows of a DTU memory section, continuous multiple-row printouts may be obtained by joining successive print messages under computer program control.

OPERATION

The operator prepares a message for printing in the same manner as for transmitting. When the print (PRT) pushbutton is depressed the mode indicator is changed to "P," and the message to be printed is transmitted to the computer. To identify the message as one to be printed, the status character for print appears in the header.

The computer buffers each print message it receives until the printer to be used is free to print the message. The computer addresses the printer to be used in the same way it addresses a DTU. The printer selection, by the computer program, should be based on the address of the DTU requesting printout. Before the computer retransmits the message to be printed to the PPC memory, the program can add labeling or other preprogrammed data to the message. At this time the computer could be used for reformatting or editing of the text to be printed.

When the PPC processes a print message, it starts with the character indicated by the entry marker position set up by the print message from the computer. From the starting position each character is taken sequentially from memory, and at the end of each line of text the PPC automatically inserts a carriage return and line feed code to provide proper operation of the printer. To facilitate linking or separating print messages, the Page Print Controller recognizes the "Blink" (BLK) character as a special command. When inserted in a message to a Page Print Controller, a single Blink character stops the printer at that point. A double Blink code causes the PPC to generate a carriage return and line feed sequence and then stop printing.

When the DATANET-760 receives a print message from the computer, it will respond with a non-text message in the same way it responds to normal information messages. For example, a print message received without error from the computer causes an "ACK" message to be returned to the computer after the message has been printed. In this way the computer program is notified of the fact that the print message was received without error and that the printer is now free to process another print message.

If a new print message is received by the PPC while it is processing the previous message, a "BUSY" message will be returned automatically to the computer. If a print message is received by the PPC when the "clear to send" line from the printer is off, a "BUSY" message will also be automatically transmitted to the computer. See Appendix F for format of message to PPC from computer.

5. ERROR CONTROL AND RECOVERY

While error conditions may be caused by many things, this section deals only with errors caused by the transmission media. Equipment malfunctions in the DATANET-760 or the computer are not considered. Since information flow between the DATANET-760 and the computer is in both directions, the detection of errors and their recovery must be performed at each end.

ERROR CONTROL AND RECOVERY AT DATANET-760

It is assumed that the operator will wait for receipt of an "ACK" message for each information message transmitted to the computer prior to transmission of a new information message.

Error conditions from which automatic recovery can be made may be caused by any of the following three conditions:

1) Absence of a transmission frame from the computer for a period in excess of the prescribed time, measured from the end of the carrier of the last transmission frame transmitted by the DLC
2) Reception of a message containing a valid address and containing a detected error
3) Reception of a message for a DTU which is busy at the time of reception

Absence of Transmission Frame

At the end of each transmission frame originated from the DLC a timer is started. If the timer should run out before a transmission frame is received from the computer, a new transmission frame will be initiated by the DLC.

Messages transmitted by the DLC and which have not been acknowledged to the operator during this period will remain in the DATANET-760 memory for available retransmission by the operator.

Reception of Message Having Error

A message containing a valid DTU address may be detected in error by any of the following ways:

1) Detected lateral parity bad in any character
2) Receipt of any status character other than ACK, NAK, or NUL after the address character
3) Receipt of any non-graphic character after the status character until the start of text character. (A non-graphic character has bit 6 and bit 7 both equal to 0.)
4) Receipt of any of the following characters after the STX and before the ETX: SOH, STX, ACK, NAK, NUL, or EOT
5) The loss of the Carrier Detected Signal from the Digital Data Set before the end of a message
6) Receipt of a new SOH after a SOH and before the ETX
7) Detected message error by an error detection scheme such as Longitudinal Parity

If a DTU which is not busy receives a message detected in error by any of the above conditions, all of the following will automatically be performed:

1) An error indication in the form of a blinking mode indicator will be displayed to the operator on the CRT display.
2) A "NAK" character will be generated in the status character position of a non-text message to be transmitted the next time the DTU is serviced by the transmit scanner.
3) The DTU keyboard which was locked during reception, is unlocked to permit continued operator usage while the "NAK" status message is transmitted back to the computer.

Reception of Message For Busy Terminal

A terminal is considered "Busy" if any of the following conditions exist at the time of decoding the status character in a received message:

1) The DTU is in the local mode
2) A transmit or print request from the keyboard has not yet been serviced
3) An acknowledgment or negative acknowledgment is awaiting transmission
4) A retransmission request has not yet been processed
5) Terminal is a page printer and a message addressed to its PPC is still being printed

However, an acknowledgment message does not cause a "BUSY" response since it does not result in conflicts.

Reception of a message for a "BUSY" terminal will cause a "BUSY" character to be generated in the status character position of a non-text message to be transmitted the next time the terminal is serviced by the transmit scanner.

Reception of Message Containing "NAK" And "ENQ"

The computer may request an automatic retransmission of the last text message received from the DATANET-760 to recover a message received in error. Such a message will contain a "NAK" status and an "ENQ" character in the text.

Upon completion of reception of a "NAK" message containing no errors and which contains an "ENQ" the following will automatically be performed:

1) The DTU keyboard will be locked until the DTU is serviced by the transmit scanner and the message is retransmitted.
2) A "NUL" character will be generated in the status character position of the message to be transmitted the next time the DTU is serviced by the transmit scanner, indicating the message contains text.
3) The text of the message to be transmitted will contain all characters between the "marker" position on the display at the beginning of the message to be transmitted and the "End of Text" symbol on the display.

Reception of Message Containing "ACK"

Reception of a message containing an "ACK" status and containing no errors for a terminal which is not busy will cause the text (if any-see footnote) of that message to be displayed to the operator as a visual acknowledgment of a previously transmitted message. No message will be generated by the DATANET-760 in response to an "ACK" message, but the mode indicator "T" or "P" will be replaced by "R" as an indication of the receipt of the "ACK."

If the operator has not waited for acknowledgment and has changed to local mode, the "L" mode character will remain on the display. The keyboard is not locked at the beginning of the message if the status character is "ACK."

Reception of Message Containing "NUL"

If a DTU which is not busy receives a "NUL" status message containing no errors, the text of the message will be displayed to the operator, and the following will be performed after reception is complete:

1) An "ACK" character will be generated in the status character position of a non-text message to be transmitted the next time the DTU is serviced by the transmit scanner.
2) The DTU keyboard will not be locked while the transmission request is present, thus allowing the operator use of the DTU.

It is recommended that text never be contained in "ACK" messages, since protection against operator interference is not provided in this case.

DATANET-760 Response Priority

When the transmit scanner services a DTU, all message requests for that DTU are serviced according to a fixed sequence before the transmit scanner advances to the next DTU. The fixed sequence is:

1) ACK message
2) NAK message
3) Retransmission
4) BSY message
5) Text message
6) PRT message

Keyboard Locking

The keyboard is electrically locked when a transmit or print request is entered from the keyboard. The keyboard is unlocked after the message is transmitted, without waiting for an acknowledgment.

The keyboard is also locked at the beginning of a received message (before the STX) unless the message is an acknowledgment. This is because acknowledgment messages do not contain text or entry marker movements and hence cannot cause conflicts with keyboard data entries. The keyboard is unlocked at the end of a received message. However, if a transmit or print request is still awaiting service, the keyboard is not unlocked.

Legal Addresses

Messages which do not contain legal addresses are ignored. For an address to be considered legal, the station bits must match the prewired station address of the DATANET-760, the specified TMU must be present, and the DTU bits must be allowable with respect to TMU partitioning.

Entry Marker Restoration

During reception of a message, the entry marker position is extracted and updated in a separate register. If a detectable error occurs, which will result in a negative acknowledgment, the updated entry marker position is not restored. Hence, a retransmission request (NAK) from the DATANET-760 can be satisfied by retransmitting the message from the computer without injection of additional entry marker positioning.

ERROR CONTROL AND RECOVERY AT THE COMPUTER

Error control and recovery at the computer is under program control, thus providing a large degree of flexibility. The following discussion describes a recommended solution; however, the inherent flexibility of the system provides wide areas for improvement and sophistication.

The computer should wait for an "ACK" message from the DATANET-760 for each information message transmitted to a particular DTU prior to transmission of a different message to that DTU.

Reception of Message Having Error

If a text ("NUL" status) message containing a valid DTU address is received in error, the computer will address a message to that DTU which will cause the error message to be automatically retransmitted from the DATANET-760. If a number of attempts (determined by the program) are not successful in obtaining a correct message, the program will transmit a message that will cause the word "ERROR" to be displayed on the first line of the DTU display, notifying the operator that his last message has not been received correctly and that manual recovery is necessary.

If a non-text ("ACK", "NAK", "BUSY" status) message containing a valid DTU address is received in error, the computer will retransmit the last message transmitted to the terminal.

Reception of Message Containing "NAK"

If a message containing a valid DTU address is received without detected error, the program should supply an automatic retransmission of the last message transmitted to that DTU.

Reception of Message Containing "BUSY"

The program will time out an appropriate time and then retransmit the last message transmitted to that DTU.

Reception of Message Containing Text (NUL)

The program will transmit an acknowledgment message to the DTU and place the DTU in the receive mode.

LONGITUDINAL PARITY OPTION

Longitudinal Parity (LP) option provides the function of longitudinal parity check and generation on each message entering and leaving the DLC.

A single LP character is generated and inserted into the message stream immediately following the ETX character of each message. It checks all characters following the SOH through the ETX.

When the LP option is not present, an equivalent character time is transmitted and must be received by the DATANET-760. Use of all ones is suggested.

The LP character consists of eight bits. The first bit of the LP character makes the total of the first bits of every character, from - but not including - SOH, even. The remaining bits of the LP character make the total of their respective bits in the preceding characters even. (i.e., the first bit of the LP checks all preceding first bits, the second all preceding second bits, the third all preceding third bits, etc.) The eighth LP bit is lateral parity for the longitudinal parity character.

6. ELECTRICAL CHARACTERISTICS, CABLING, AND ENVIRONMENT

POWER REQUIREMENTS

Display Controller Unit

The power source required is 105 to 125 volts, 60 cycles single phase. Power consumption will be from 350 watts to 750 watts, depending on controller configuration.

Display Terminal Unit

The power source required is 105 to 125 volts, 60 cycles single phase. Power consumption is 200 watts.

UNDERWRITERS LABORATORY REQUIREMENTS

All materials meet or exceed UL standards for fire and shock hazards. Where UL listings are available on material and/or components, the listed items are used in preference to non-listed ones.

RADIO FREQUENCY INTERFERENCE

The DATANET-760 meets the applicable FCC specification, part 15, Incidental and Restricted Radiation Devices, for suppression of radio frequency interference.

CABLING

Information transfer between a DTU and the DCU is accomplished by use of two coaxial cables. One cable is used for transmission of video information to the DTU, and one cable for the transmission of keyboard data from the DTU to the DCU.

For cabling distance of 1000 feet or less between DTU and DCU, a direct connection by coaxial cable is made. The cable type required between the DCU and the DTU for distances of 1000 feet or less is RG59B/U (0.242 inch O.D.).

Up to four receive-only monitors may be connected on the video distribution cable, in addition to the DTU. The receive-only monitors are connected in series from a jack at the rear of the active DTU. A switchable termination resistor is provided on each receive-only monitor. The termination resistor must be switched in at the last monitor in the chain.

ENVIRONMENTAL CONDITIONS

The DCU will operate in normal office and factory environments which meet the following conditions:

Room ambient: 65°F (18°C) to 85°F (30°C)
Room humidity: 20% to 80% relative
No condensed moisture or extremely dust laden or corrosive atmosphere.

The DTU will operate under the following conditions:

Room ambient: 40°F (4°C) to 100°F (38°C)
Room humidity: 10% to 90% relative
No condensed moisture or extremely dust laden or corrosive atmosphere.

APPENDIX A
COMMUNICATION LINE INTERFACE SIGNALS AND PIN ASSIGNMENTS

PIN	FUNCTION
1	Protective Ground
2	Transmitted Data
3	Received Data
4	Request to Send
5	Clear to Send
6	LINE ON
7	Signal Ground
8	Data Carrier Detector
9	SOURCE ?
10	
11	
12	
13	
14	
15	Transmitter Signal Element Timing*
16	
17	Receiver Signal Element Timing*
18	
19	
20	Data Terminal Ready
21	
22	PHONE RINGING
23	
24	
25	

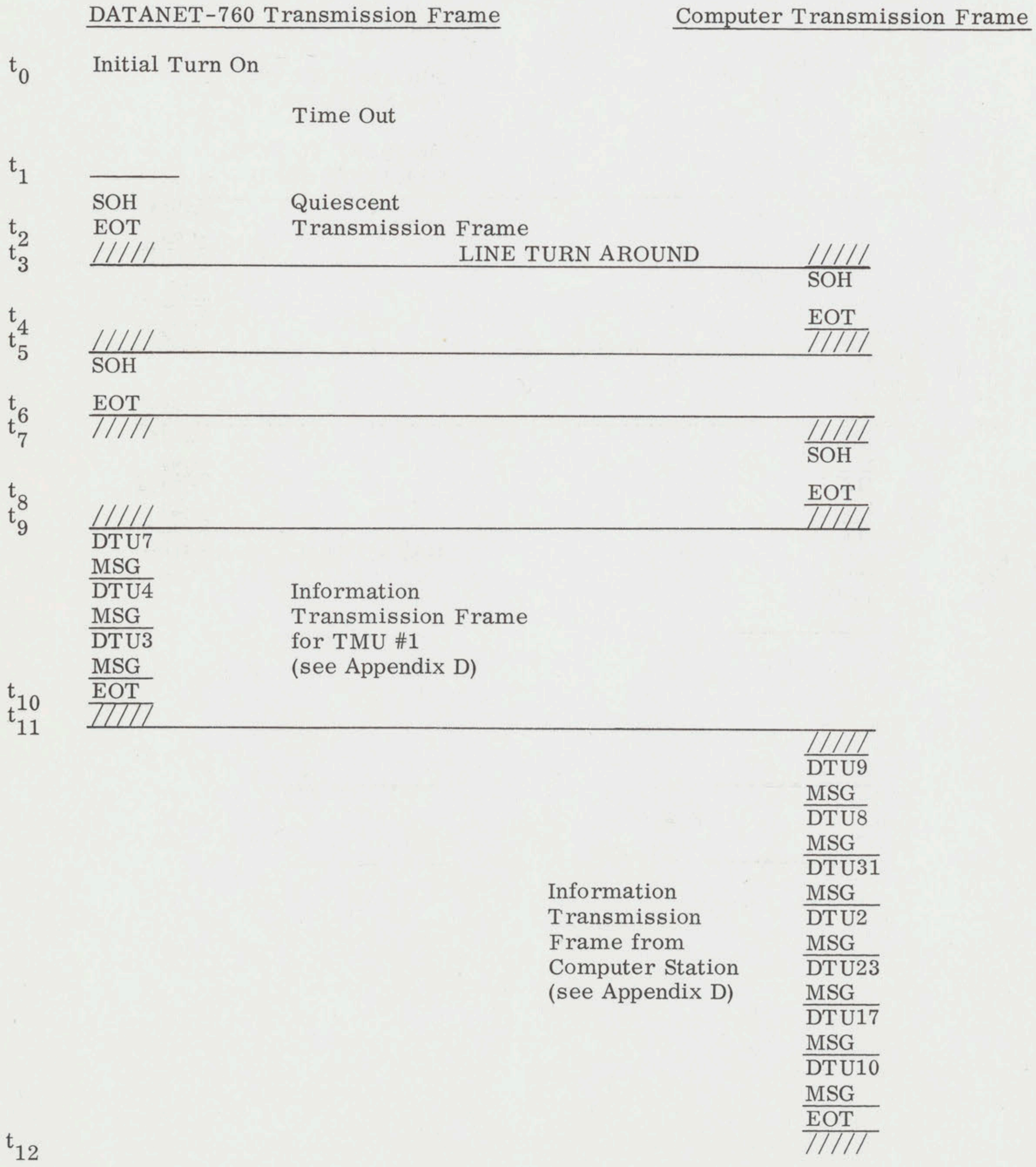
*2400 Baud Synchronous Data Transmission Only

APPENDIX B
PAGE PRINT CONTROLLER SIGNALS

PIN	SIGNAL
1	PROTECTIVE GND
2	TRANSMITTED DATA
3	
4	REQUEST TO SEND
5	CLEAR TO SEND
6	
7	SIGNAL GND
8	
9	
10	
11	
12	
13	
14	
15	
16	
17	
18	
19	
20	DATA TERMINAL READY
21	
22	
23	
24	
25	

APPENDIX C

TYPICAL HALF DUPLEX TRANSMISSION FRAME SEQUENCE



t_{12}	/////	
t_{13}	DTU24	
	MSG	
	DTU18	
	MSG	
	DTU17	
	MSG	
t_{14}	EOT	
t_{15}	/////	/////
		DTU3
		MSG
		DTU7
		MSG
t_{16}		EOT
t_{17}	/////	/////
	SOH	
t_{18}	EOT	
t_{19}	/////	/////
		DTU4
		MSG
		DTU24
		MSG
		EOT
t_{20}		/////

APPENDIX D
INFORMATION TRANSMISSION FRAMES

From DATANET-760

```
Syn*
Syn*
Syn*
Syn*
SOH
ADR (DTU 7)
STAT (NUL in text message)
FC1 }
FC2 } space codes -- reserved for future use.  Always transmitted by DATANET-760.
STX
Text
ETX
LP    (Longitudinal Parity - all ones if not present)
Syn*
Syn*
Syn*
Syn*
SOH
ADR (DTU 4)
STAT
FC1
FC2
STX
Text
ETX
LP
Syn*
Syn*
Syn*
Syn*
SOH              }
ADR (DTU 3)      }
STAT             } Message Format (from DATANET-760)
FC1              }
FC2              }
STX              }
Text
ETX
LP
EOT
```

*Syn used only with synchronous DLC's.

From Computer

(same as from DATANET-760 with the exception of the "FC1, FC2" space codes - which need not be sent)

APPENDIX E
MESSAGE FORMAT FOR REQUEST FOR RETRANSMISSION OF MESSAGE RECEIVED IN ERROR BY THE COMPUTER

SOH

Display Terminal Address

NAK

STX

PR - (This may be any other sequence of marker control characters which will move the marker on the CRT back to the beginning of the message to be retransmitted.)

ENQ

ETX

If a number of attempts (determined by the program) were not successful in obtaining a correct message, the program may transmit an error message with the following content:

SOH

Display Terminal Address

NUL

STX

PR

E

R

R

O

R

ETX

APPENDIX F

FORMAT FOR MESSAGE FROM COMPUTER TO PAGE PRINT CONTROLLER

SOH

PPC ADDRESS — Selects which printer is to be used. The fact that a message is addressed to a PPC indicates that the message is to be printed.

NUL

STX

FORM FEED — This character is not stored in the PPC memory but at the time it is received, it clears the PPC memory of any previous message data and starts the new entry at the first character position in memory. This character would not be present if a portion of the previous print message is to be retained and reprinted.

T
E — to be printed
X
T

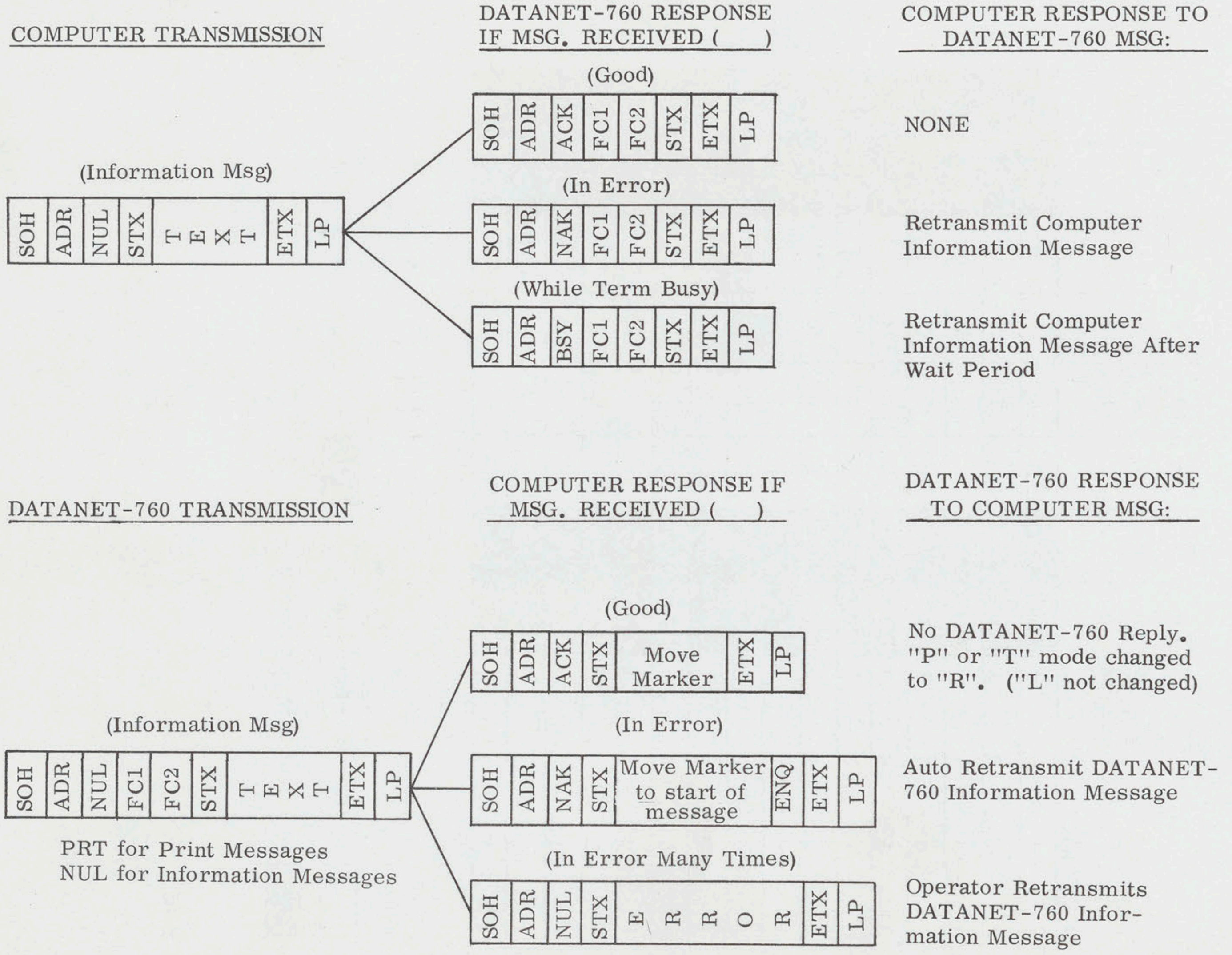
BLINK — When received, this character is stored in the PPC memory as a normal data character. When the PPC is transferring data to the printer set and this character is read from memory, it is an indication to the PPC to stop printing action with no further motion of the print carriage. Two blinks in succession stop printing and cause a carriage return and line feed. If desired, the program may format the print message with no blink code for full page printouts. In this case, after the last character position of the memory is printed, carriage return and line feed codes are generated to the printer and then printing stops. Using this capability, a continuous printout of many lines of text can be programmed with a series of print messages. For example, those print messages in the series which are full page messages would have no blink codes inserted. Those which have less than a full page of text lines would have a blink code in the first character position on the line following the last line of text, or two blinks in succession after the last character to be printed on the last line. Thus, each message in the series would always end with a carriage return and line feed. The next message would then start at the beginning of the text line following the last line of the preceding message.

PAGE RETURN — This character is not stored in memory when received. It starts printout with the first character position in memory. Printout can start with other than the first character if desired by use of a suitable entry marker position sequence.

ETX

APPENDIX G

SUMMARY OF MESSAGE TYPES AND REQUIRED RESPONSES



COMPUTER TRANSMISSION	DATANET-760 RESPONSE IF MSG. RECEIVED ( )	COMPUTER RESPONSE TO DATANET-760 MSG:
(Information Msg) SOH ADR NUL STX TEXT ETX LP	(Good) SOH ADR ACK FC1 FC2 STX ETX LP	NONE
	(In Error) SOH ADR NAK FC1 FC2 STX ETX LP	Retransmit Computer Information Message
	(While Term Busy) SOH ADR BSY FC1 FC2 STX ETX LP	Retransmit Computer Information Message After Wait Period

DATANET-760 TRANSMISSION	COMPUTER RESPONSE IF MSG. RECEIVED ( )	DATANET-760 RESPONSE TO COMPUTER MSG:
(Information Msg) SOH ADR NUL FC1 FC2 STX TEXT ETX LP PRT for Print Messages NUL for Information Messages	(Good) SOH ADR ACK STX Move Marker ETX LP	No DATANET-760 Reply. "P" or "T" mode changed to "R". ("L" not changed)
	(In Error) SOH ADR NAK STX Move Marker to start of message ENQ ETX LP	Auto Retransmit DATANET-760 Information Message
	(In Error Many Times) SOH ADR NUL STX ERROR ETX LP	Operator Retransmits DATANET-760 Information Message

APPENDIX H
CHARACTER SET FORMAT

			b7	0	0	0	0	1	1	1	1
			b6	0	0	1	1	0	0	1	1
			b5	0	1	0	1	0	1	0	1
b4	b3	b2	b1								
0	0	0	0	NUL	▨	SP	0	@	P	▨	▨
0	0	0	1	SOH	RLF	!	1	A	Q	▨	▨
0	0	1	0	STX	FS	"	2	B	R	▨	▨
0	0	1	1	ETX	▨	#	3	C	S	▨	▨
0	1	0	0	EOT	PR	$	4	D	T	▨	▨
0	1	0	1	ENQ	NAK	%	5	E	U	▨	▨
0	1	1	0	ACK	SYN	&	6	F	V	▨	▨
0	1	1	1	▨	▨	'	7	G	W	▨	▨
1	0	0	0	BS	▨	(	8	H	X	▨	▨
1	0	0	1	▨	▨	)	9	I	Y	▨	▨
1	0	1	0	LF	PRT	*	:	J	Z	▨	▨
1	0	1	1	▨	▨	+	;	K	|	▨	▨
1	1	0	0	FF	▨	,	<	L	—	▨	▨
1	1	0	1	CR	▨	-	=	M	|	▨	▨
1	1	1	0	▨	▨	.	>	N	BLINK	▨	▨
1	1	1	1	▨	▨	/	?	O	┌	BUSY	▨

□ - Character for Storage

▨ - Combinations not interpreted

▒ - Commands (not stored)

CHARACTER SET LEGEND

NUL - Null. Used as a status character in message header to indicate the message contains text.

SOH - Start of Header

STX - Start of Text

ETX - End of Text

EOT - End of Transmission

ENQ - Enquiry. Used in conjunction with NAK as subsequently described.

ACK - Acknowledgment. Used as a status character to indicate that the last message for the specified terminal was received correctly.

BSY - Busy. Used as a DATANET-760 status character in the message header to indicate that the last message received was addressed to a Busy terminal. A terminal is Busy when it is under local control or has a message waiting to be transmitted.

PRT - Print. Used as a DATANET-760 status character to indicate that the message is to be printed. The computer then re-addresses the message to the PPC.

NAK - Negative Acknowledgment. Used as a status character in message header to indicate that the last message was received with a detectable error.

BS - Backspace marker one character. When the marker is at the left edge of the page, subsequent backspaces have no effect.

LF - Line Feed. Moves marker down one line. When the marker is on the bottom line, a line feed moves the marker to the top line without changing character position.

FF - Form Feed. Clears memory, erases display, and page returns marker to the top left character position.

CR - Carriage Return. Returns marker to the leftmost character position. When the marker is at the left edge of the page, subsequent carriage returns have no effect.

RLF - Reverse Line Feed. Moves marker up one line without changing character position. When marker is on top line, subsequent RLF's have no effect on the marker. Character position is never changed with this command.

FS - Forward Space marker one character. When the marker reaches the end of the line, another Forward Space causes generation of a Carriage Return - Line Feed.

PR - Page Return. Returns marker to top left character position without erasing display.

CHARACTER SET LEGEND (CONT)

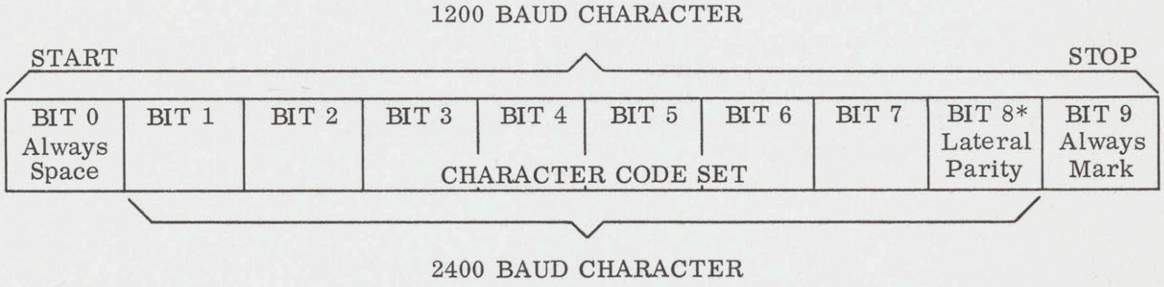
SYN - Synchronous Idle. Used in synchronous transmission to obtain character synchronization.

SP - Space. Stores and displays a "Space" character.

BLINK - Causes all characters between Blink character and the next space character (or end of line) to Blink. When inserted in a message to a Page Print Controller, a single Blink character stops the printer at that point. A Double Blink code causes the PPC to generate a carriage return, line feed sequence and then stop printing.

APPENDIX J

GENERAL CHARACTER COMPOSITION



BIT 0	BIT 1	BIT 2	BIT 3	BIT 4	BIT 5	BIT 6	BIT 7	BIT 8*	BIT 9
Always Space			CHARACTER CODE SET					Lateral Parity	Always Mark

* Even for 1200 Baud, Odd for 2400 Baud.

ADDRESS (ADR) AND STATUS (STAT) CHARACTER COMPOSITION:

BIT 1	BIT 2	BIT 3	BIT 4	BIT 5	BIT 6	BIT 7	MEANING
ADDRESS CHARACTER:							
D	D	D	T	T	S	S	DDD are DTU designation bits. Up to 8 DTU's are allowed per TMU. TT are TMU designation bits. Up to 4 TMU's are allowed within a DATANET-760. SS are Station Designation bits. Station Bits of DATANET-760 are factory wired as 11.
STATUS CHARACTERS:							
0	0	0	0	0	0	0	NUL. Message contains text
1	1	1	1	0	1	1	BSY. Terminal was busy when last computer message was received. (Valid in DLC originated messages only.)
0	1	1	0	0	0	0	ACK. Last message was received correctly.
1	0	1	0	1	0	0	NAK. Last message was received incorrectly.
0	1	0	1	1	0	0	PRT. Message is to be printed. (Valid only in DLC originated messages.)

All other combinations not used

APPENDIX K

ALLOWABLE DISPLAY TERMINAL ADDRESS BITS VERSUS TMU PARTITIONING

TMU Partitioning	Allowable DTU Address Bits (b3, b2, b1)	
1 DT (26 Lines)	000	(DT1)
2 DT's (16 Lines each)	000	(DT1)
	100	(DT2)
4 DT's (8 Lines each)	000	(DT1)
	010	(DT2)
	100	(DT3)
	110	(DT4)
8 DT's (4 Lines each)	000	(DT1)
	001	(DT2)
	010	(DT3)
	011	(DT4)
	100	(DT5)
	101	(DT6)
	110	(DT7)
	111	(DT8)

INFORMATION SYSTEMS DIVISION

GENERAL ELECTRIC

LITHO IN U.S.A.

Jack:
If interested let me know.
Gene

CC: MGR. ____________ SALES DEVELOPMENT

REQUEST FOR NON-STANDARD PRICE QUOTATION

DATE 21 August 1967

TO: MANAGER, PRODUCT PLANNING

CUSTOMER: Artificial Intelligence Group - MIT

SPECIAL PRODUCT NAME: New D/N-760 K/B and Display Terminal Unit

PURPOSE: To replace the standard keyboard and display terminal unit that is concurrently in use with the D/N-760 & PDP-6 located in the Artificial Intelligence Group's laboratory at MIT.

DESCRIPTION:
(Use additional pages as required. Reference any attachments. Include block diagrams if appropriate.)

Solid State keyboard and accompanying display terminal unit. (This is the new keyboard which is being developed for the D/N-760 and is presumably available 1st quarter '68.

DATE QUOTATION REQUIRED: ____________

An order can be expected for this non-standard product if the monthly rental does not exceed $__________ or the sale price does not exceed $__________.

REQUESTED BY: Dominic G. Talone   SALES REPRESENTATIVE, DIAL COMM 8-262-8127

APPROVAL: ____________   DISTRICT MANAGER   DATE ____________

APPROVAL: ____________   REGIONAL MANAGER   DATE ____________

FOR PRODUCT PLANNING USE   IDENT. NO. ____________

DATE RECEIVED: ____________   DISPOSITION:

CC: MGR. ______________ SALES DEVELOPMENT

REQUEST FOR NON-STANDARD PRICE QUOTATION

DATE 21 August 1967

TO: MANAGER, PRODUCT PLANNING

CUSTOMER: Artificial Intelligence Group - MIT

SPECIAL PRODUCT NAME: Data Line Controller (DLC-5) for D/N-760

PURPOSE: Present bit transmission rate of the DLC located at the AIGs laboratory is 2400 bps. Customer has requested on many occasions that the bit rates be increased.

DESCRIPTION:
(Use additional pages as required. Reference any attachments. Include block diagrams if appropriate.)

The new DLC-5 with a 50,000 BPS rate could be incorporated by using the same terminal memory units & basic controller. All changes would be made on the back panel wiring. It is necessary for Phoenix to permit Oklahoma City engineering to release presently available DLC-5 performance specifications.

DATE QUOTATION REQUIRED: ______________

An order can be expected for this non-standard product if the monthly rental does not exceed $__________ or the sale price does not exceed $__________.

REQUESTED BY: Dominic G. Talone

SALES REPRESENTATIVE, DIAL COMM 8* 262-8127

APPROVAL: ______________ DISTRICT MANAGER DATE ______________

APPROVAL: ______________ REGIONAL MANAGER DATE ______________

FOR PRODUCT PLANNING USE
DATE RECEIVED: ______________

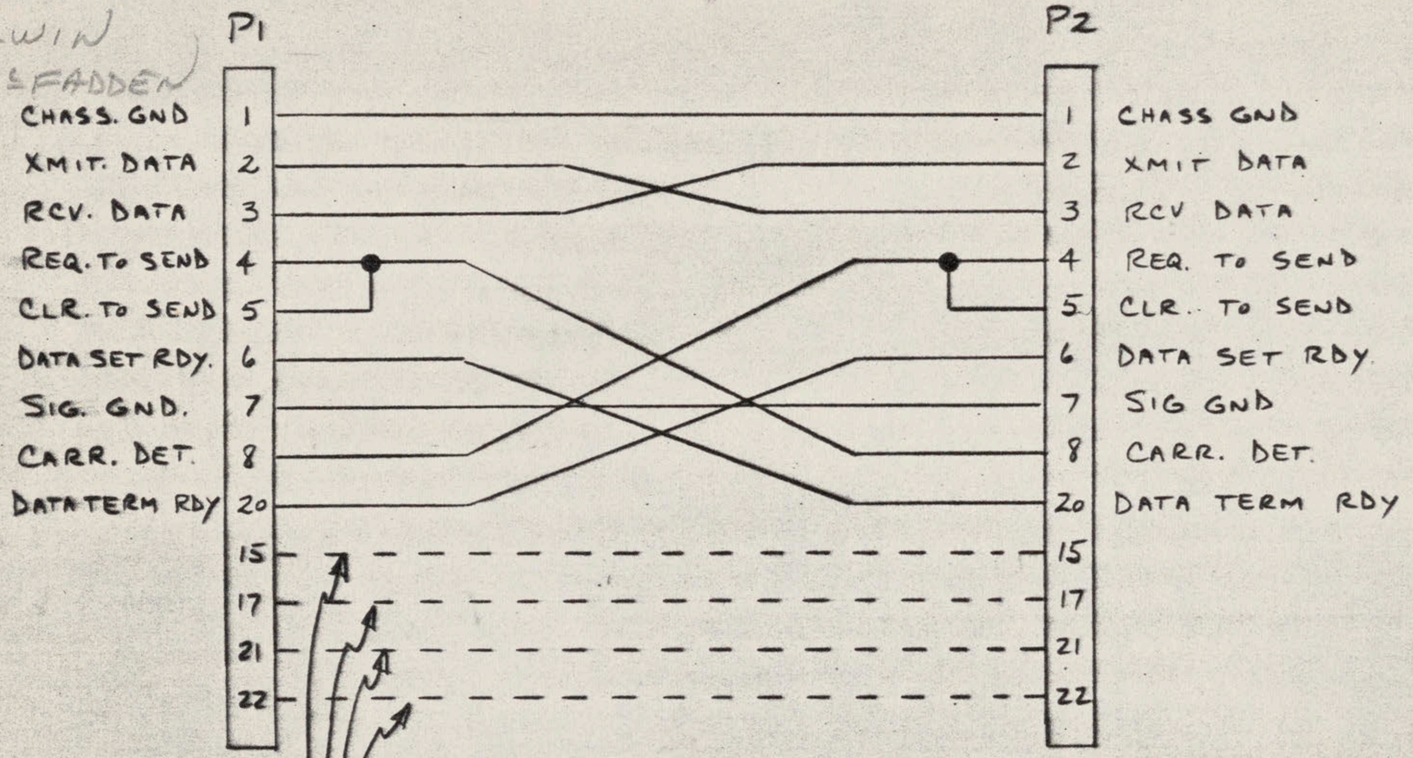
IDENT. NO. ______________
DISPOSITION:

8-353-2316

W 8

JOHN IRWIN
GEORGE McFADDEN

CONNECTION DIAGRAM:



WIRES 15, 17, 21, AND 22 ARE OPTIONAL, FOR USE WHEN TIE POINTS FOR ALL CONDUCTORS OF 12 COND. CABLE ARE NEEDED

ACE"
IES
E -
ED
SET.

C

UNLESS OTHERWISE SPECIFIED DIMENSIONS ARE IN INCHES. TOLERANCES ON:	SIGNATURES	DAY	MO	YR
FRACTIONS ± DECIMALS ± ANGLES ±	DRAWN J. M. IRWIN	11	4	66
	CHECKED			
	ISSUED			
	ENGRG			
ALL SURFACES ✓	MFG			
MATL-	MATLS			

GENERAL ELECTRIC
DEPT LOC

ADAPTER CABLE
DATANET 760 DLCI TO
DATANET 30/COMPUTER - DIRECT TIE

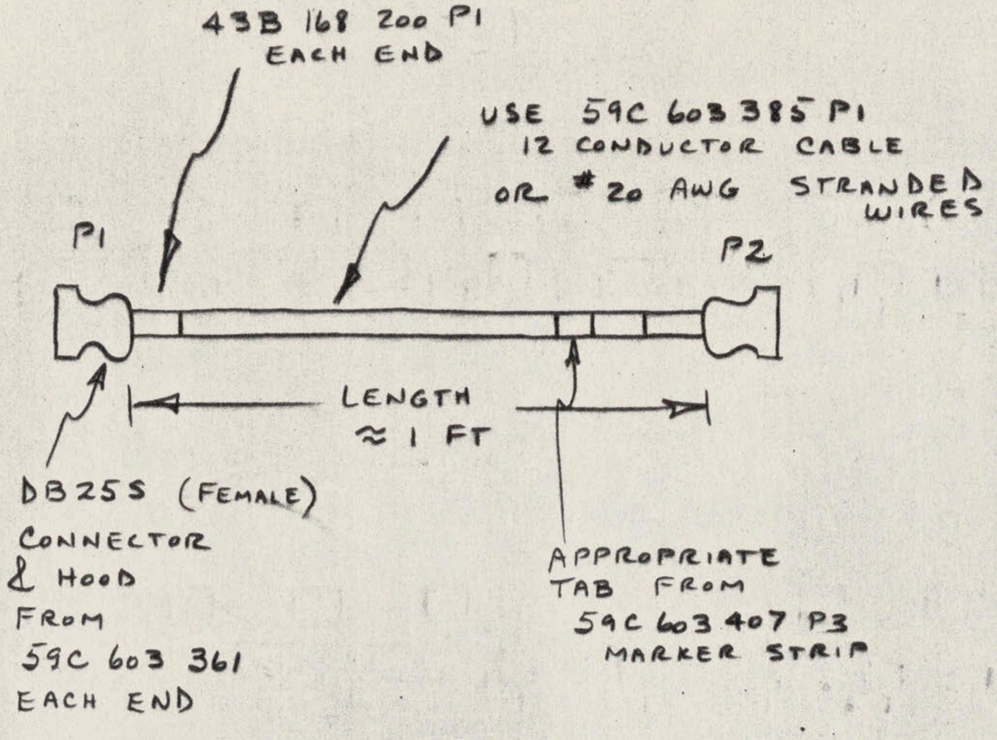
SIZE B	CODE IDENT NO.	
SCALE		SHEET

DIST TO

SOH SOH

SOH ADR SOH

↑



CHASS. GN
XMIT. DA
RCV. DA
REQ. TO S
CLR. TO S
DAT ET
SIG. GN
CARR. DE
DATA TERM

NOTES: 1. CABLE IS SYMMETRICAL; HENCE P1 AND P2 MAY BE INTERCHANGED.

2. CABLE MAY ALSO BE USED FOR DIRECT CONNECTION OF DATANET 760 PPC TO TELETYPEWRITER IF TTY IS EQUIPPED WITH "EIA INTERFACE" CONNECTOR CORD. THIS APPLIES ONLY TO PURCHASED TELETYPEWRITERS, SINCE TTY's LEASED FROM BELL SYSTEM USUALLY CONTAIN INTEGRAL 103G DATA SET.

7 6 5 4 1

1 1 |TAU 4
ADR| |0 0 0|

TA

00 –	A
01	B
10	C
11	D

```
TECO-
TECO
300,ØEC$$
$$

3$










1$
TEC

TECO
3,ØEC$$
3EK3↑U$$
3,ØEC$$
3EK3↑U$$
$$

L$DDT
3$

D$1$

D$DDT
G$
34000$
D$DDT
G$
100!   MOVEI 1,1000
101!   CONO 750,200
102!   CONSO 750,100
103!   JRST .-1
104!   MOVE 2,(1)
105!   JUMPL 2,ZZ#
106!   DATAO 750,2
107!   JRST 102
110!   ZZ:   MOVEI 1,1000
ZZ+1!   JRST 102
```

1067
1057

```
1000!    1 .
1001!    160XXX   160
1002!    0
1003!    2.
1004!    106
1005!    11XXX   OXXX   117
1006!    117
1007!    3
1010!
```

SPACE	100	1	2	4	10	20	40
	1	2	3	4	5	6	7

3

1

1011111

0000011
0100000
0110001
0110001
0110001
1100000
0110010
ϕ
1000000

1000001
1010000
1011000
1011000
1011000
1110000
1011001
1100000
1000001

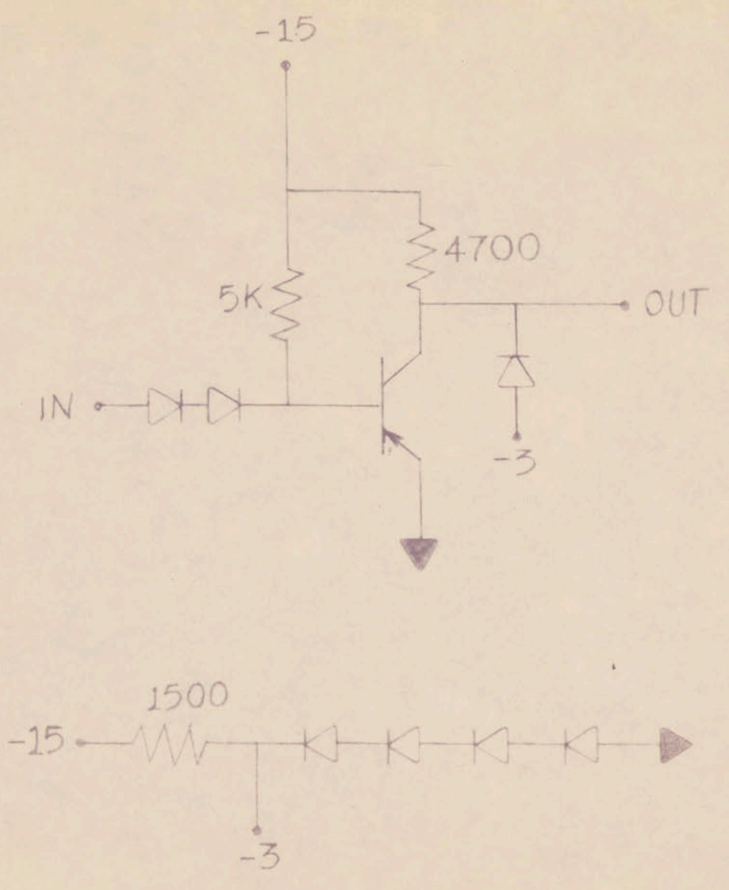
1
300
2

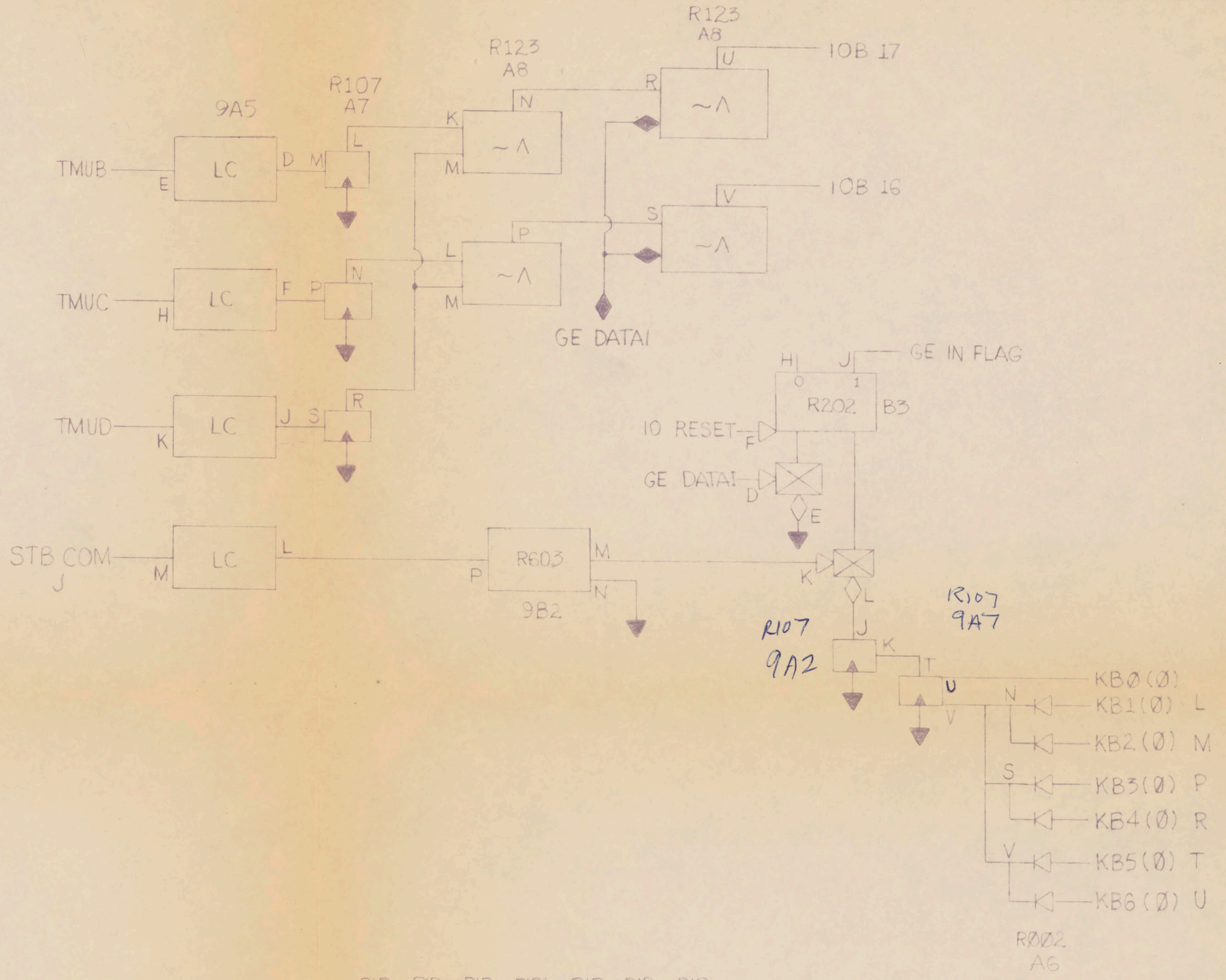
100
60
1
47
43
101
23
0
-1

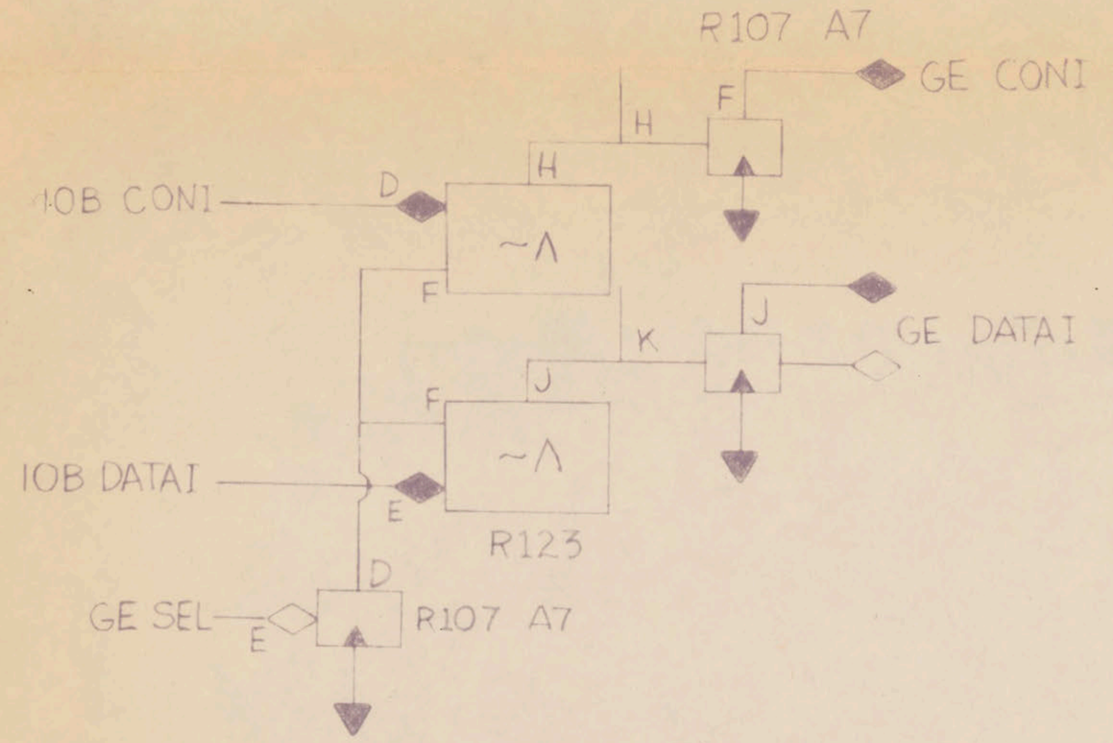
1
140
2
106
106
106
3
46
ϕ

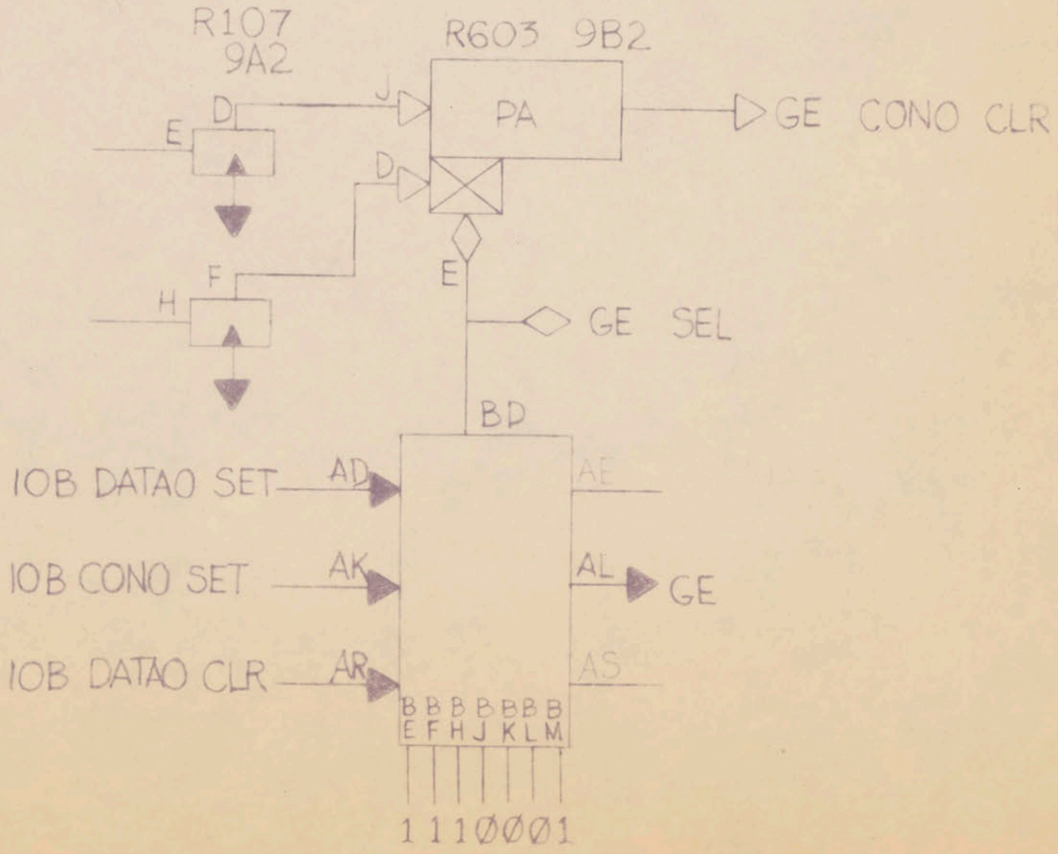
DATE	SYM	REVISION RECORD	AUTH.	DR.	CK.

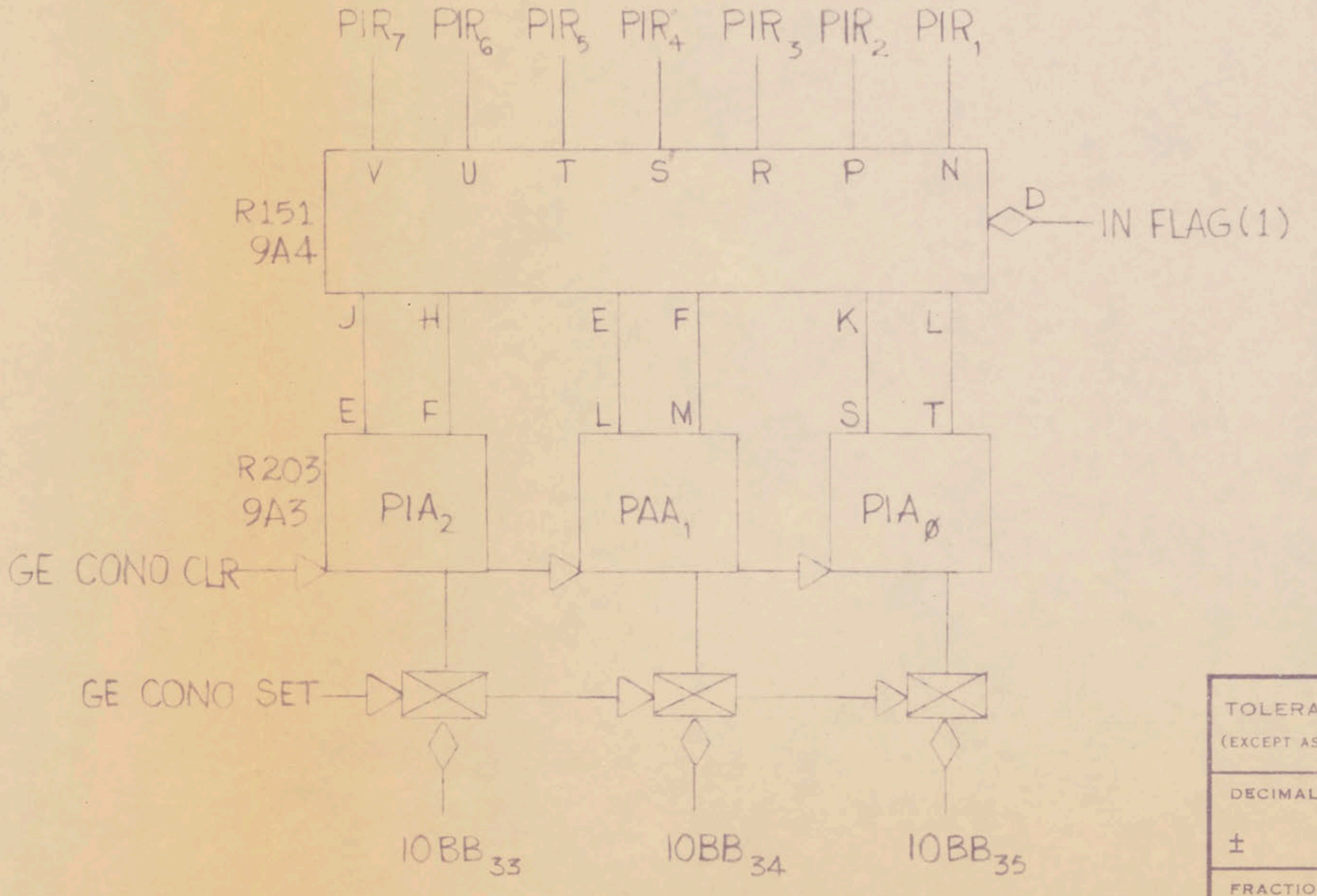
A	NC
B	-15
C	GND
D	1-OUT
E	1-IN
F	2-OUT
H	2-IN
J	3-OUT
K	3-IN
L	4-OUT
M	4-IN
N	5-OUT
P	5-IN
R	6-OUT
S	6-IN
T	NC
U	NC
V	NC





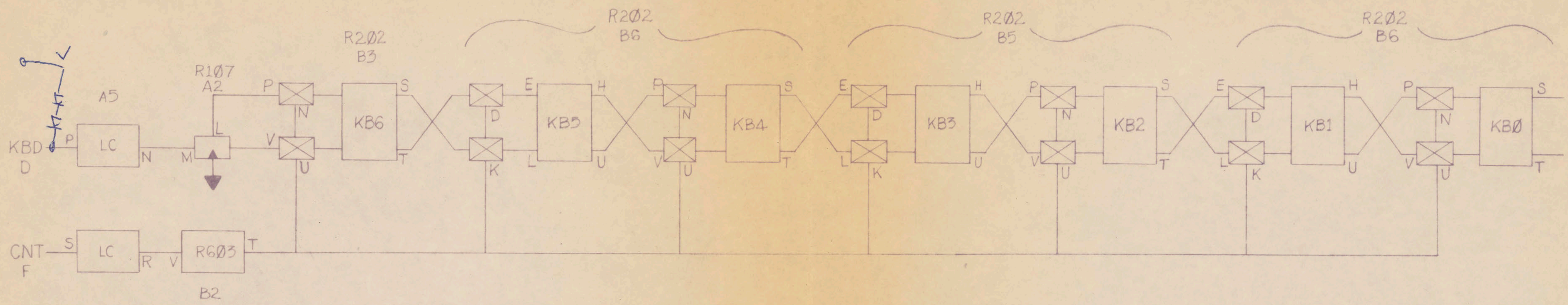


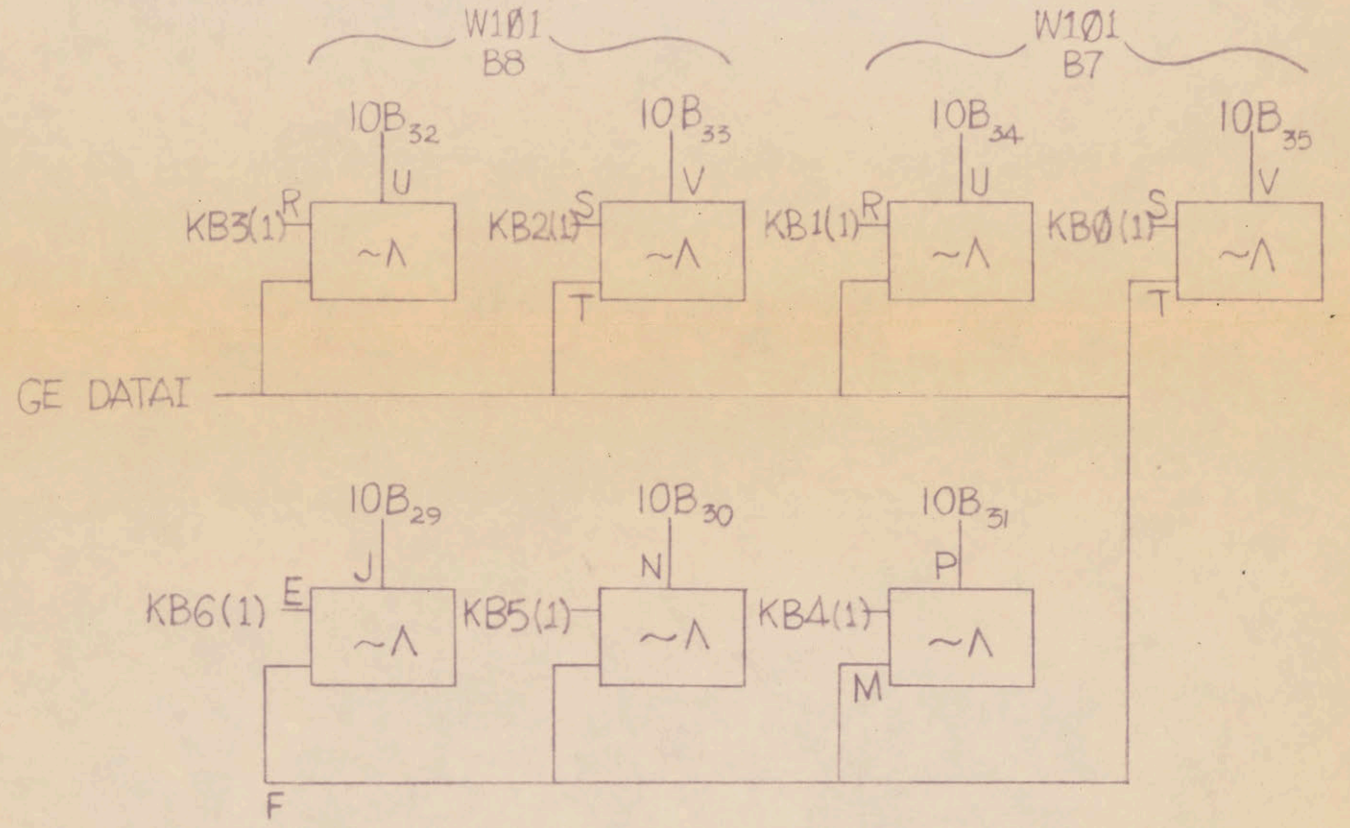


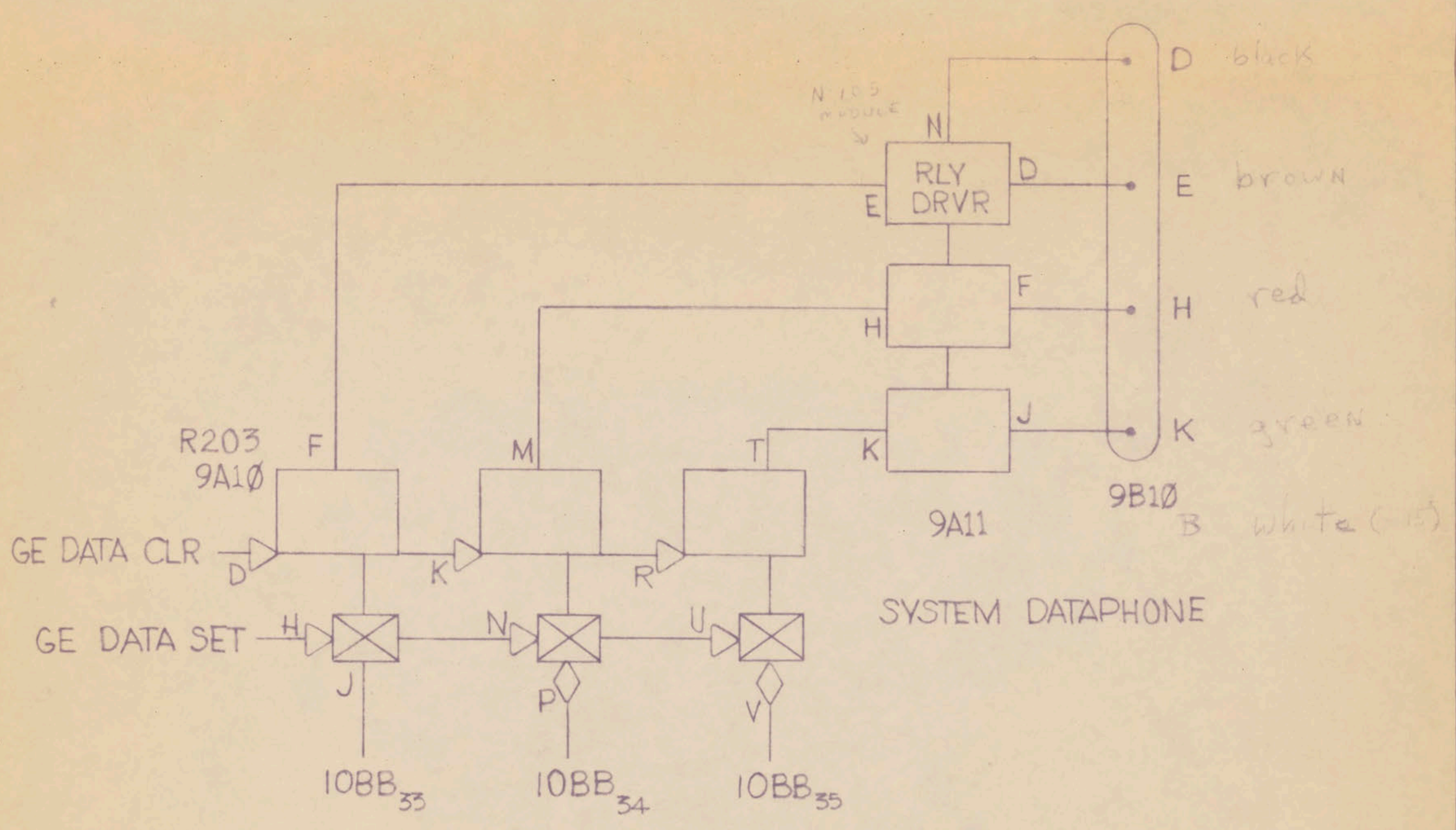


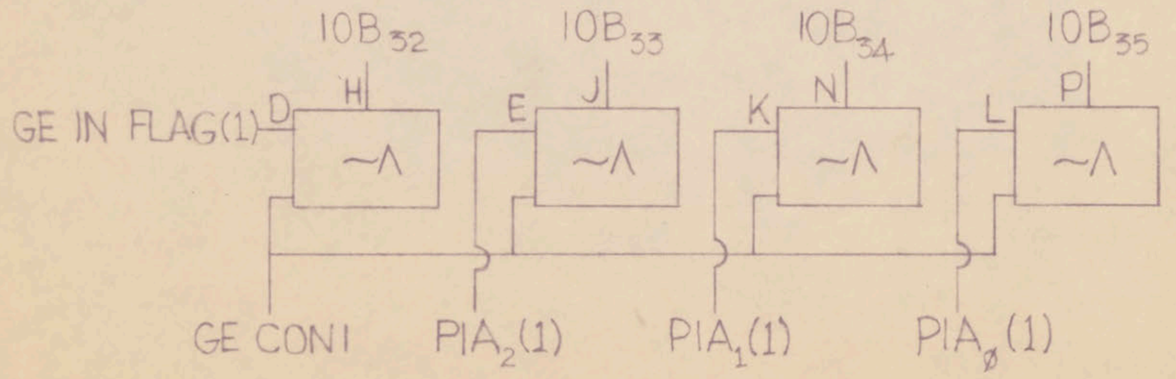
TOLERANCES (EXCEPT AS NOTED)		SCALE	DRAWN BY
DECIMAL ±			APPROVED BY
FRACTIONAL ±	TITLE		
ANGULAR ±	DATE	DRAWING NUMBER	

DATE	SYM	REVISION RECORD	AUTH.	DR.	CK.









TOLERANCES
(EXCEPT AS NOTED)

DECIMAL
±

FRACTIONAL
±

ANGULAR
±

SCALE

DRAWN BY

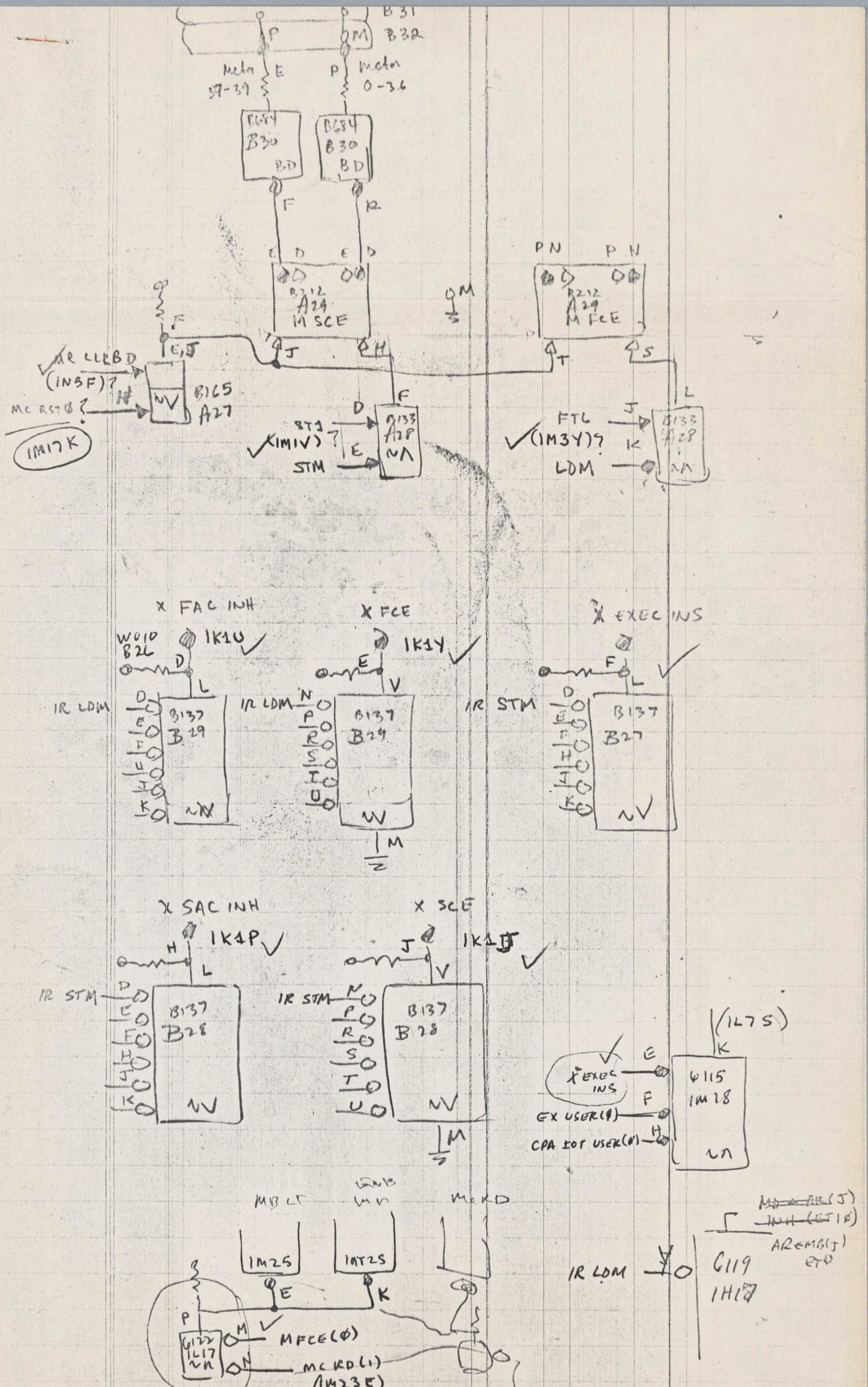
APPROVED BY

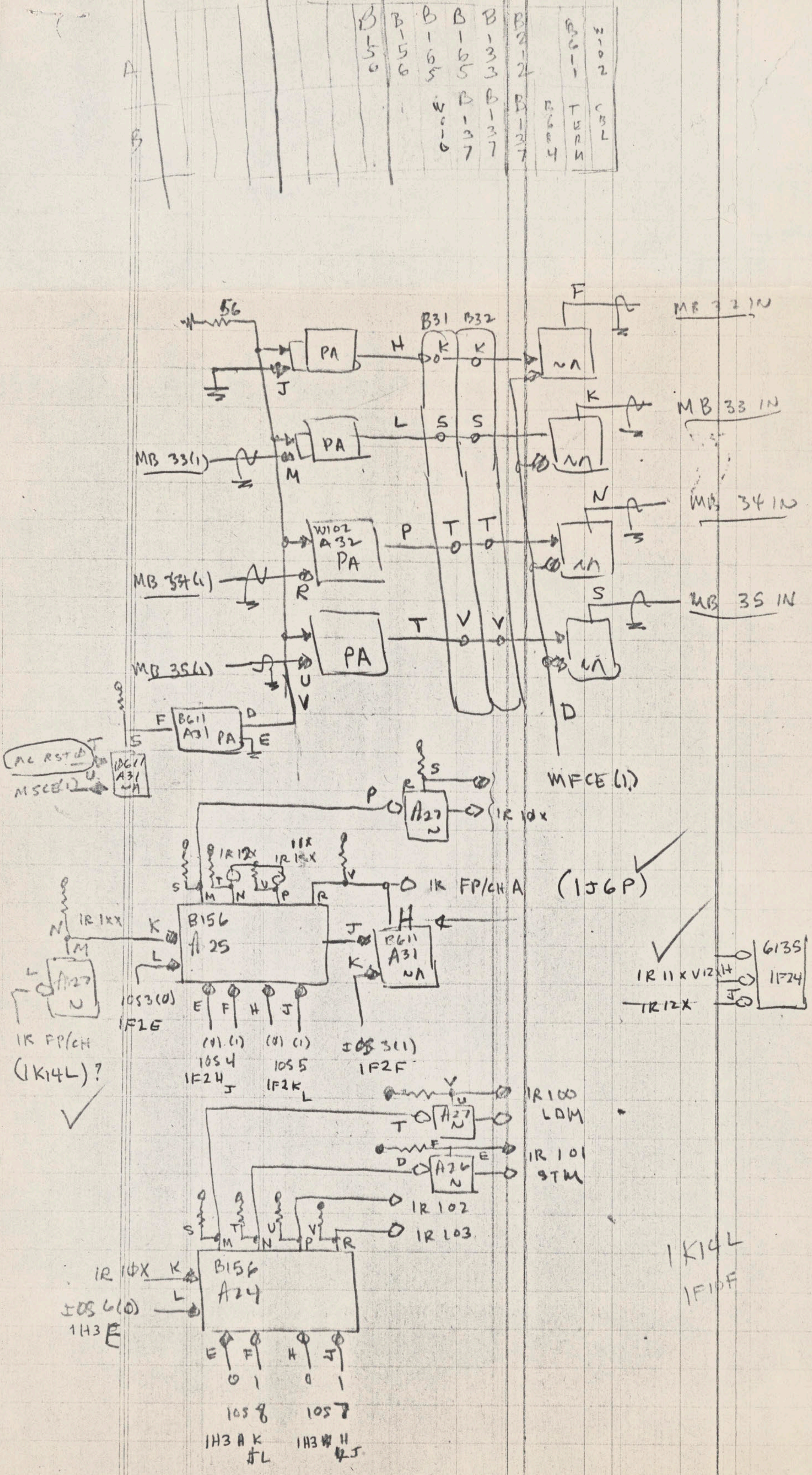
TITLE

DATE

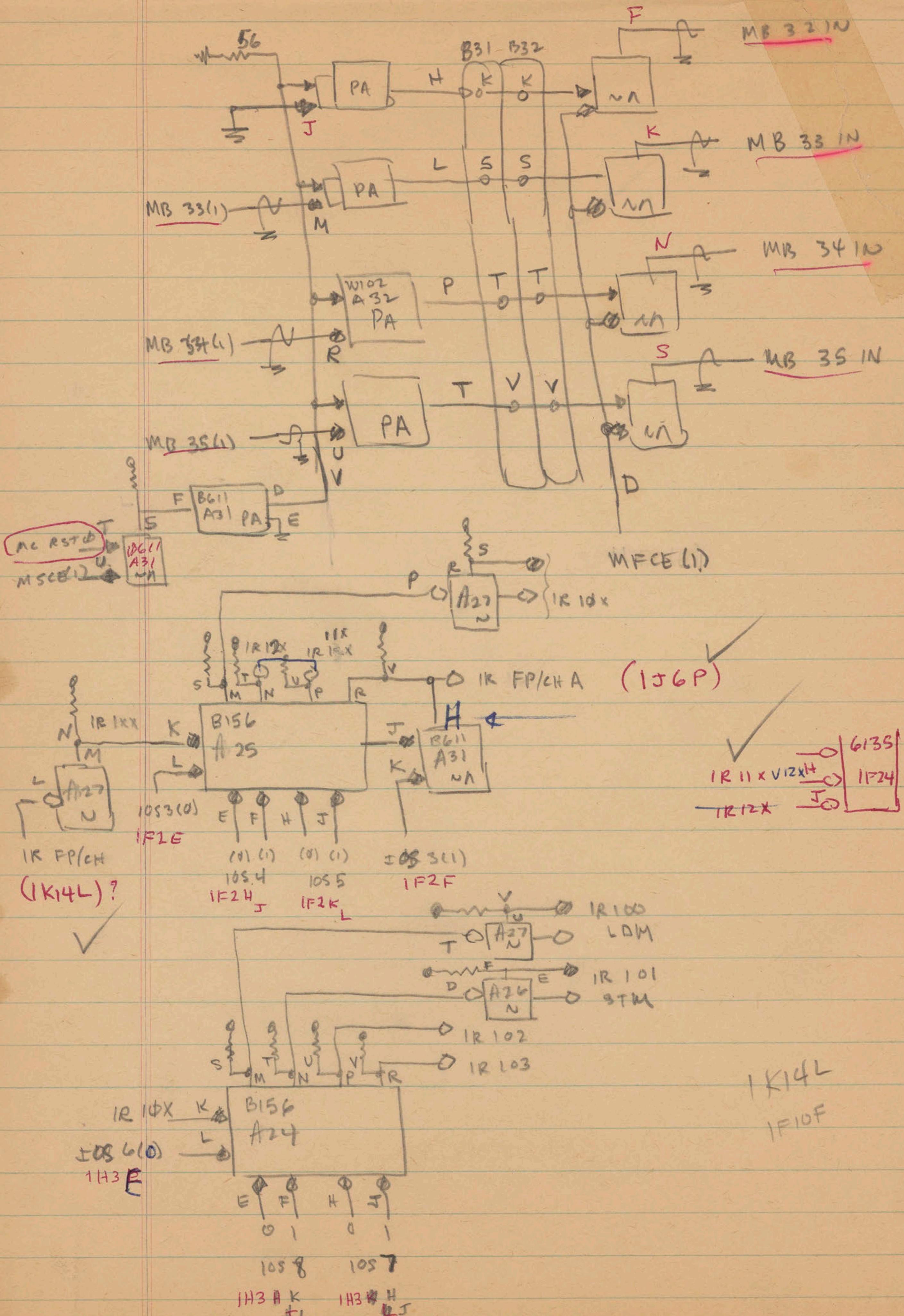
DRAWING NUMBER

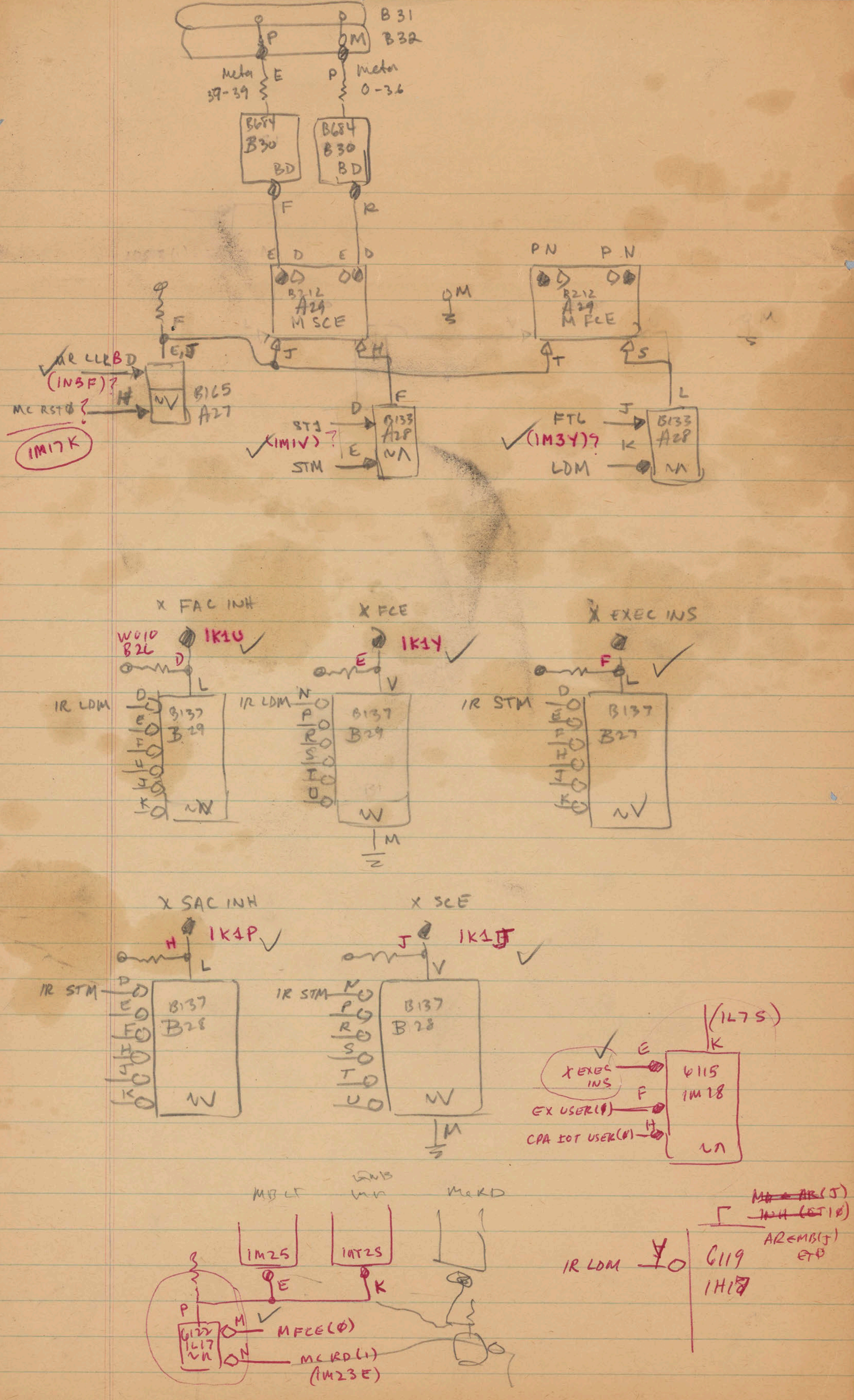
meta BITS


B31
P
M
B32
Meta
E
37-39
P
Meta
0-3.6
BG84
B30
BD
BG84
B30
BD
F
R
E D E D
B212
A29
M SCE
OM
PN PN
B212
A29
M FCE
F
E,J
J
H
T
S
AR CLLBD
(IN5F)?
MC RST0?
H
NV
B165
A27
(IM17K)
D
F
ST1
(IM1V)?
E
STM
B133
A28
NA
L
FTL
(IM3Y)?
J
K
LDM
B133
A28
NA
X FAC INH
W010
B26
D
IK1U
L
IR LDM
D
E
F
H
J
K
B137
B29
NV
X FCE
E
IK1Y
V
IR LDM
N
P
R
S
T
U
B137
B29
NV
M
X EXEC INS
F
L
IR STM
D
E
F
H
J
K
B137
B27
NV
X SAC INH
H
IK1P
L
IR STM
D
E
F
H
J
K
B137
B28
NV
X SCE
J
IK1J
V
IR STM
N
P
R
S
T
U
B137
B28
NV
M
(IL7S)
K
X EXEC
INS
E
F
EX USER(1)
H
CPA IOT USER(0)
G115
IM18
NA
MB LF
MV
MLRD
IM25
IM25
E
K
~~MA AR(J)~~
~~INH (CJ10)~~
AR←MB(J)
ETC
IR LDM
G119
IH17
P
M
N
G122
IL17
NA
MFCE(0)
MCRD(1)
(IM23E)

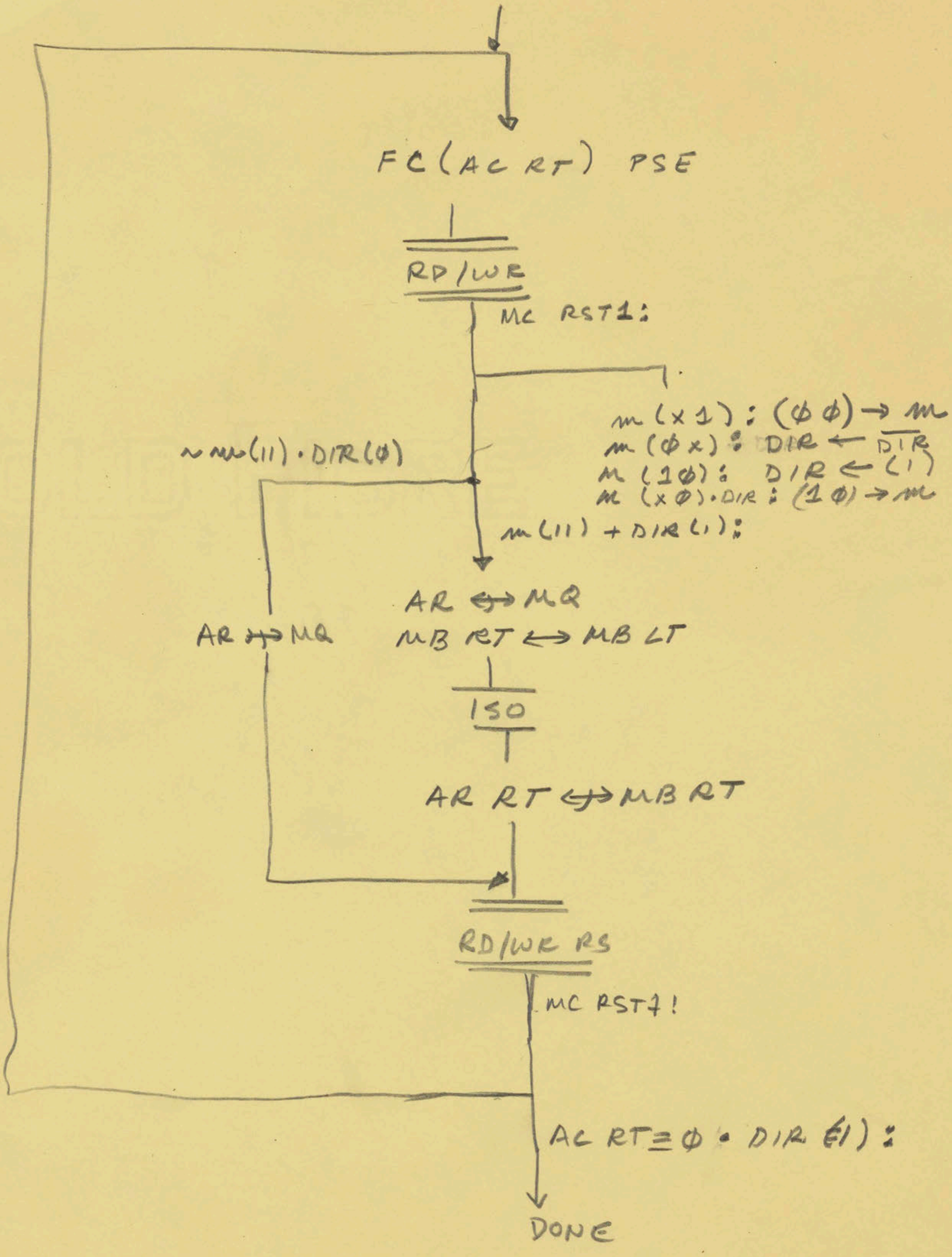

A
B
B156
B156
B165 W010
B165 B137
B133 B137
B212 B137
B611 B684
B611 TERM
W102 CBL
56
B31 B32
PA
H
K K
J
F
~A
MB 32 IN
L
S S
PA
MB 33(1)
M
K
~A
MB 33 IN
W102 A32 PA
P
T T
MB 34(1)
R
N
~A
MB 34 IN
PA
T
V V
MB 35(1)
U
V
S
~A
MB 35 IN
D
F
B611 A31 PA
D
E
S
AC RST(1)
T
U
B611 A31 ~A
MSCE(1)
MFCE(1)
S
R
P
A27 N
IR 10x
S
M
N
IR 12X
T
U
11X
IR 11X
P
V
R
IR FP/CH A
(1J6P)
✓
N
IR 1XX
M
K
B156 A25
L
J
B611 A31 ~A
H
K
A27 N
L
1OS3(0)
1F2E
E F H J
(0) (1) (0) (1)
1OS4
1F24 J
1OS5
1F2K L
IOS 3(1)
1F2F
IR FP/CH
(1K14L)?
✓
✓
IR 11 x V12X
H
J
IR12X
613S
1F24
V
U
IR 100
LDM
T
A27 N
F
E
IR 101
STM
D
A26 N
IR 102
IR 103
S
T
U
V
M
N
P
R
IR 10X
K
B156 A24
L
IOS 6(0)
1H3 E
E F H J
0 1 0 1
1OS 8
1OS 7
1H3 A K
8L
1H3 H
4J
1K14L
1F10F

A	B156	B156	B165	B165	B133	B212		B611	W102
B			W010	B137	B137	B137	B684	TERN	CBL




B31
B32
P
M
Meter 37-39
E
P
Meter 0-36
B684 B30 BD
B684 B30 BD
F
R
E D E D
B212 A29 M SCE
P N P N
B212 A29 M FCE
M
F
E,J
J
H
T
S
MR CLR BD
(IN3F)?
H
MC RST0 ?
B165 A27
IM17K
F
D
ST1
(IM1V) ?
E
STM
B133 A28
L
FTL
(IM3Y)?
J
K
LDM
B133 A28
X FAC INH
W010 B26
IK1U
D
L
IR LDM
D E F H J K
B137 B29
X FCE
IK1Y
E
V
IR LDM
N P R S T U
B137 B29
M
X EXEC INS
F
L
IR STM
D E F H J K
B137 B27
X SAC INH
H
IK1P
L
IR STM
D E F H J K
B137 B28
X SCE
J
IK1T
V
IR STM
N P R S T U
B137 B28
M
(IL75)
K
E
X EXEC INS
6115 1M18
F
EX USER(1)
H
CPA IOT USER(0)
MB LF
GNB
MR
MC RD
IM25
IAT25
E
K
P
M
6122 IL17
MFCE(0)
N
MC RD(1)
(IM23E)
~~MA ← AR(J)~~
~~INH (ST10)~~
AR←MB(J)
ET0
IR LDM
Y
6119
1H18

11	UNMARKED FREE
Ø1	UNMARKED FUll
ØØ	MARKED FUll or ½ marked Free
10	marked Free



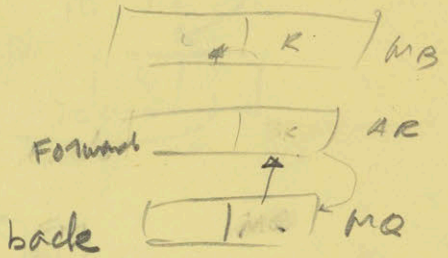
11 UNMarked Free

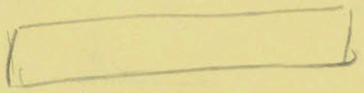
ɸ1 UNMARKED Full

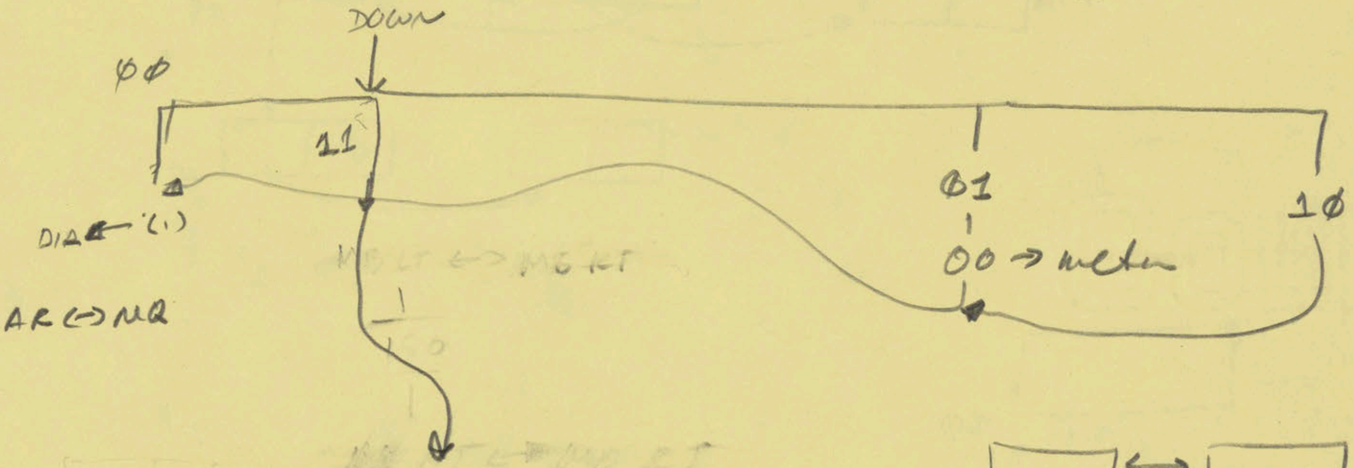
00 ½ Marked Free + Marked Full

10 Marked Free

11 + DIR(1)







00 → Meta

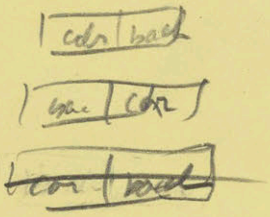
MB RT → AR RT

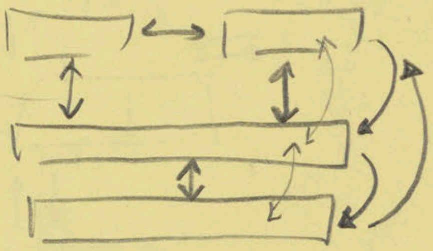
AR RT → MQ RT

MQ RT → MB RT

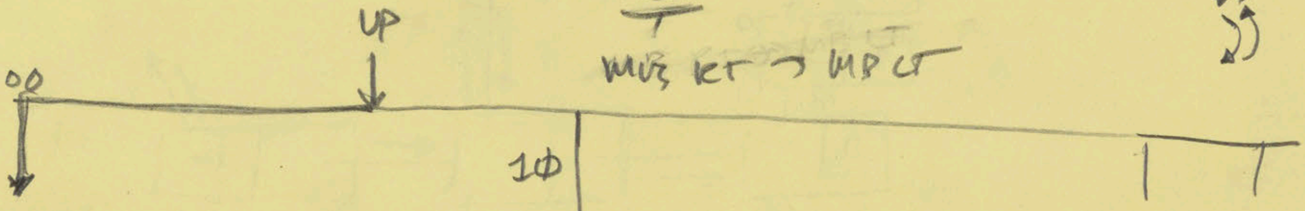
150

MB RT → MB LT





MB RT↔AR RT



MQ RT → MB RT

MB RT → AR RT

AR RT → MQ

1ɸ → Meta

1 ɸ → DIR

150

MB RT↔MB LT

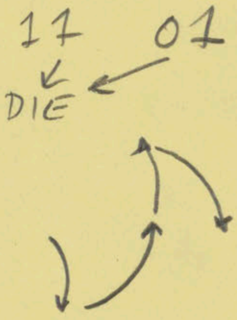
MQ RT → MB

MB RT → AR

AR RT → MQ

150

MB RT → MB LT

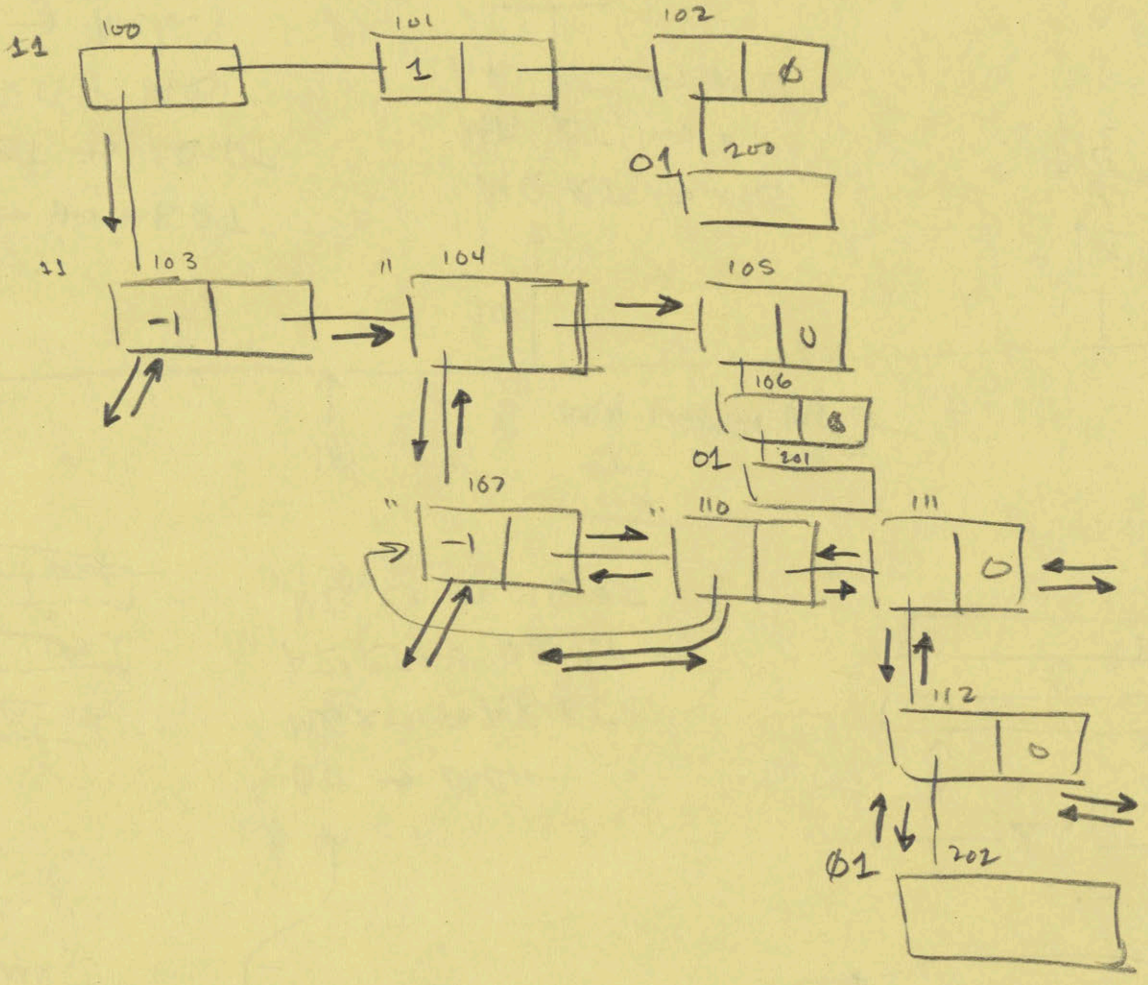


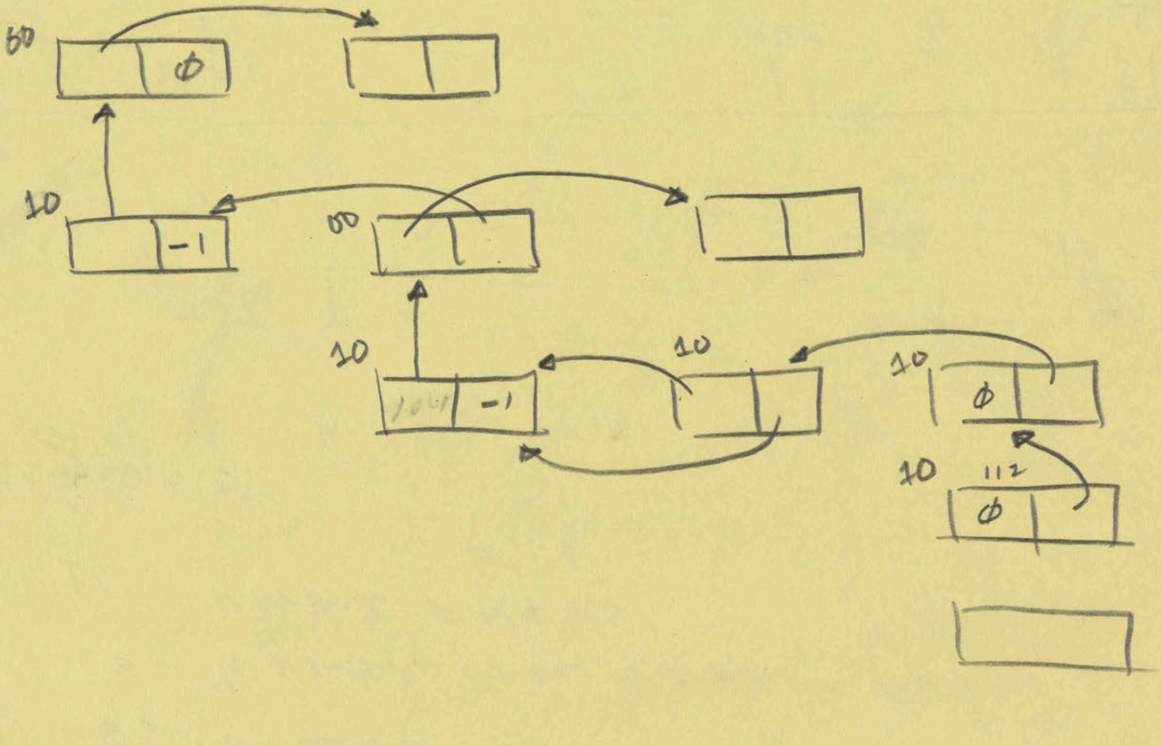
MQ↔AR

AR RT ↔ MB

AC – 100
AC+1 – ∅
DIR – ∅

(FOO ∅ –1)





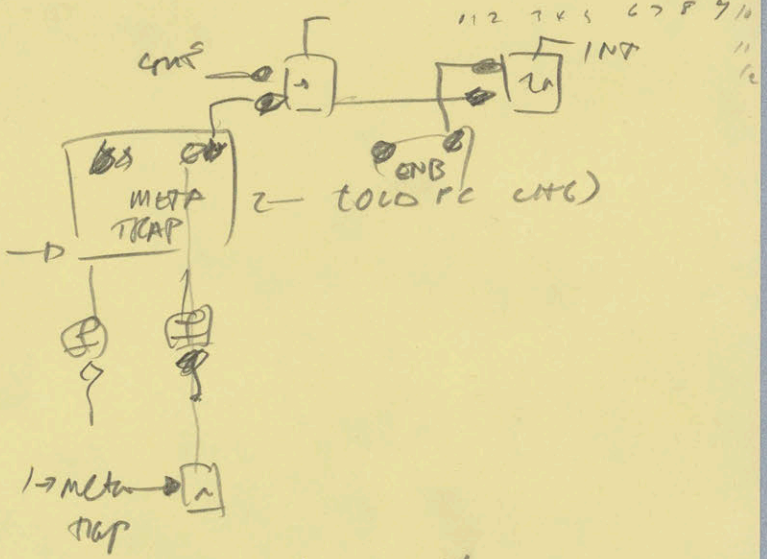
					AR	MQ
					100	ϕ
↓	100	00	103	101	0	100
			101	ϕ	103	100
	103	00	-1	104	100	103
			104	100	-1	103
↑	-1	00	→		103	-1
↓	103	10	104	100	-1	103
			100	-1	104	103
	104	00	107	105	103	104
			105	103	107	104
	107	00	-1	110	104	107
			110	104	-1	107
↑	-1	00			107	-1
↓	107	10	110	104	-1	107
			104	-1	110	107
	110	00	107	111	107	110
			111	107	107	110
↑	107	10			110	107
↓	110	10	111	107	107	110
			107	107	111	110
	111	00	112	ϕ	110	111
			0	110	112	111
	112	00	202	ϕ	111	112
			ϕ	111	202	112
↑	202	01 00			112	202
↓	112	00 10	ϕ	111	202	112
			111	202	ϕ	112
↑	ϕ	00			112	ϕ

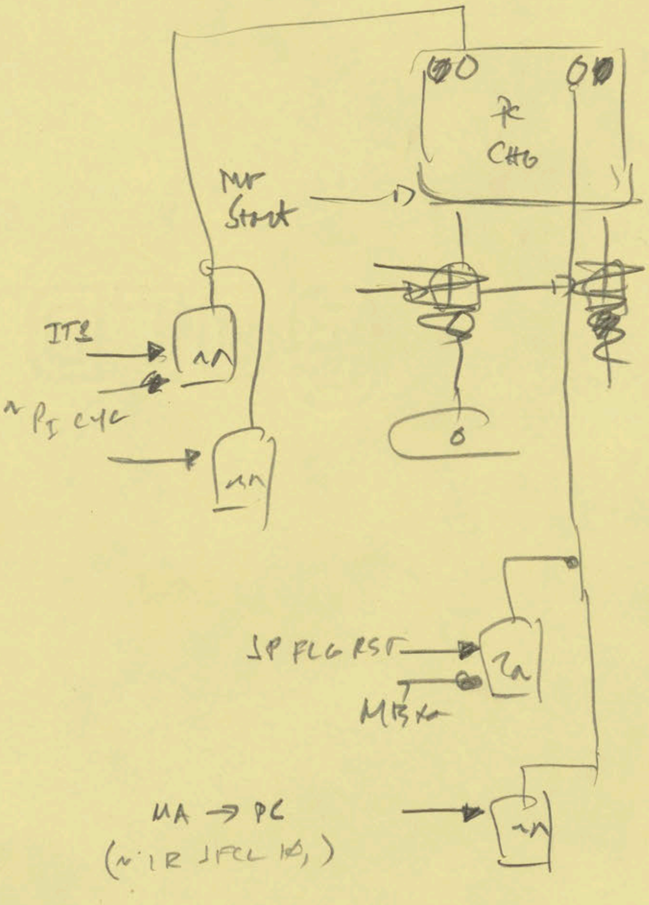
↑	112	10	111	202	ϕ	112
			202	ϕ	111	112
↓	111	00 10	ϕ	110	112	111
			110	112	ϕ	111
↑	ϕ	00	—		111	ϕ
↑	111	10	110	112	ϕ	111
			112	ϕ	110	111
↑	110	10	107	107	111	110
			107	111	107	
	107	10	104	-1	110	107
			-1	110	104	
↓	104	00 10	105	103	107	104
			103	107	105	

105 etc

PC CHANGE / META

	at IT1		at mem strb
	XCT	JMP	WRT
ØØ	ok	ok	ok
Ø1	no	no	ok
1Ø	ok	ok	no
11	ok	no	no



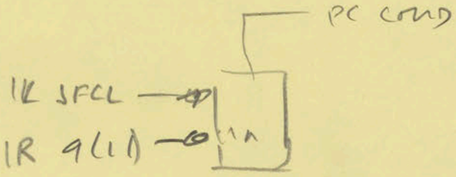


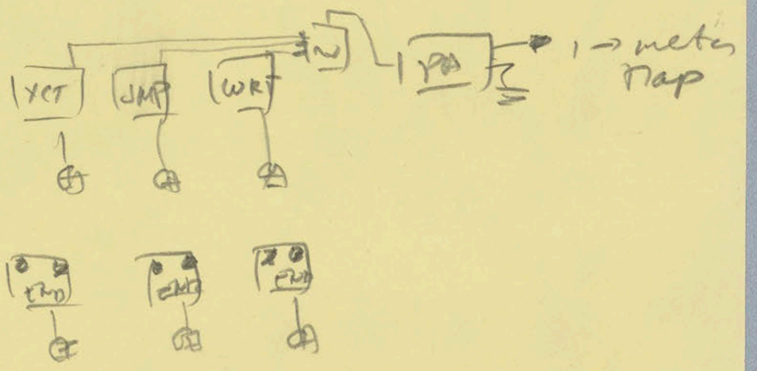
JFCL 10

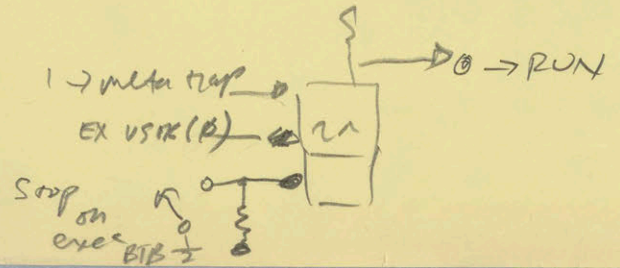
UNCONDITIONAL JMP

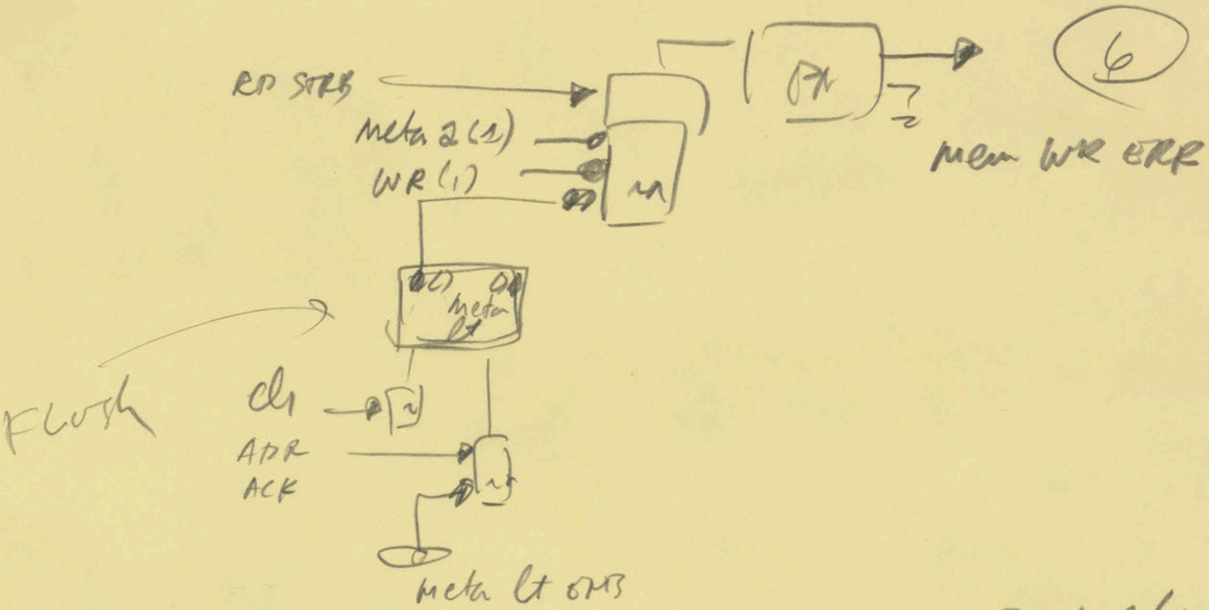
1. DDT Breakpoints
2. MAKE ILL MEM REF DO THE RIGHT THING (PUSHJ).

JRST
JRA
JSP
PUSHJ
POPJ
JFCL
JSA
JSR



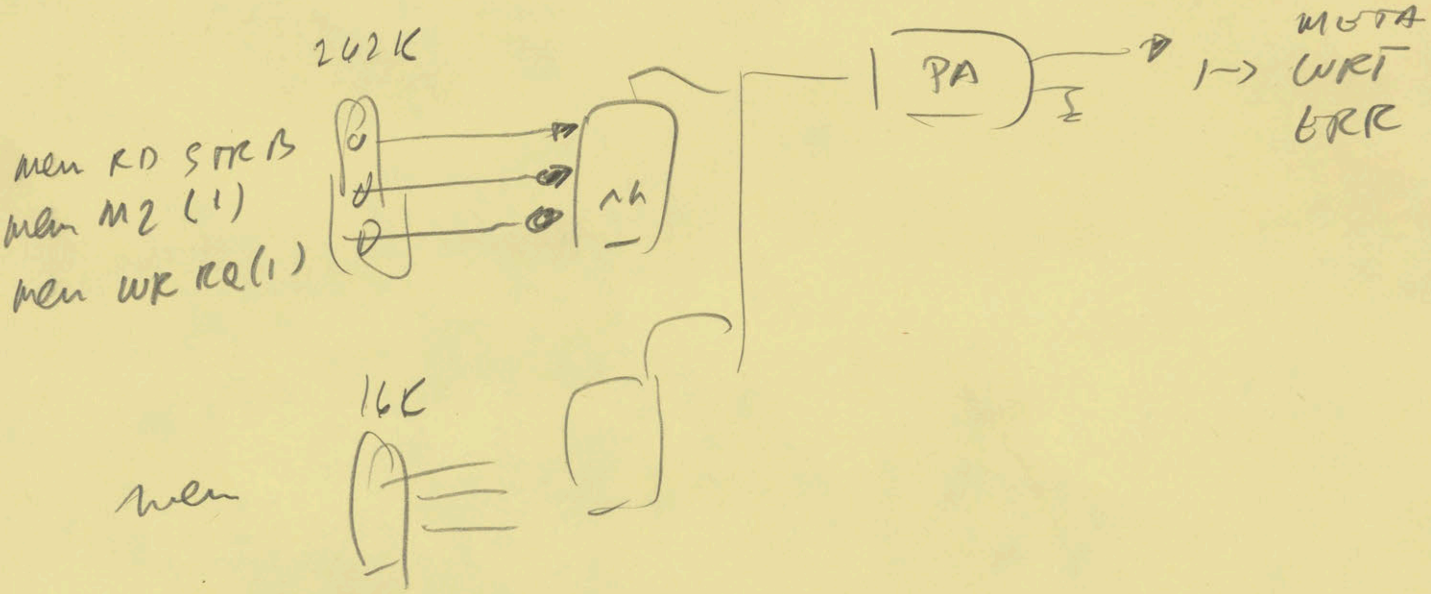


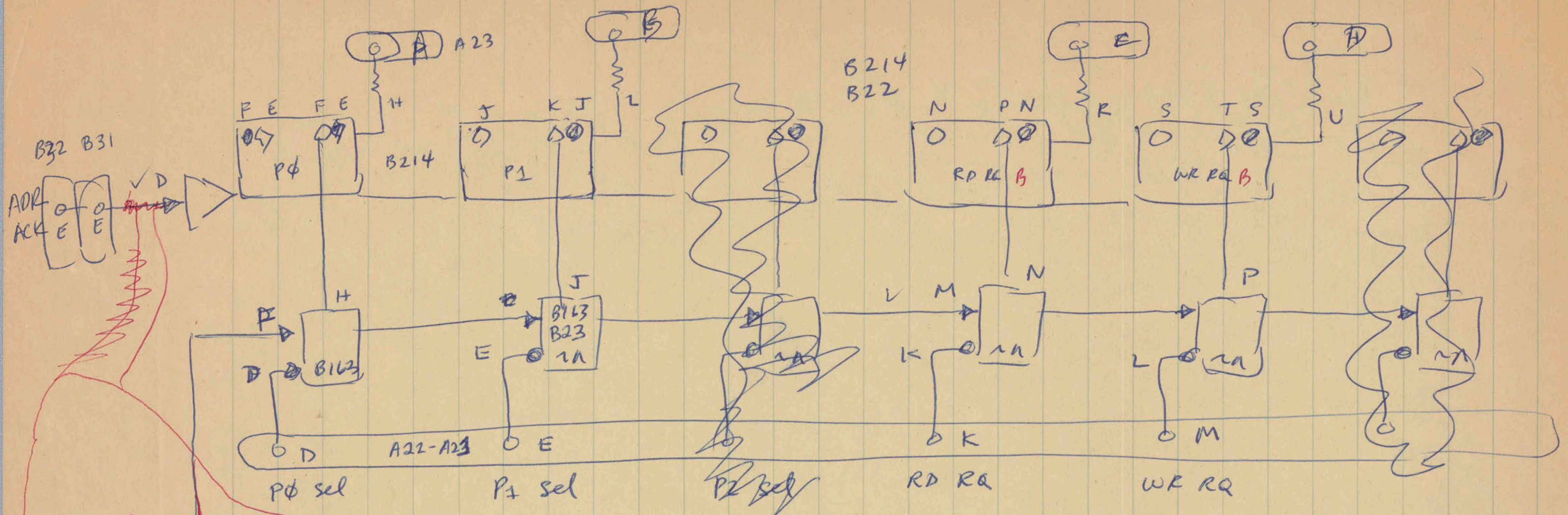


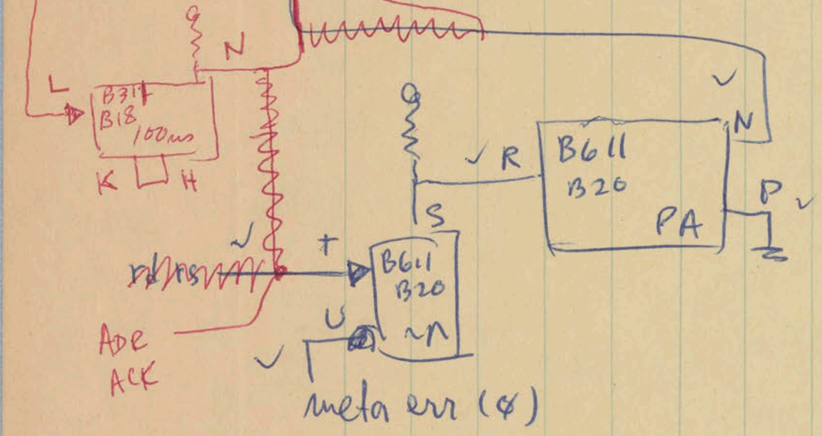


1. wr err on meta 2 detected at memory read strobe.

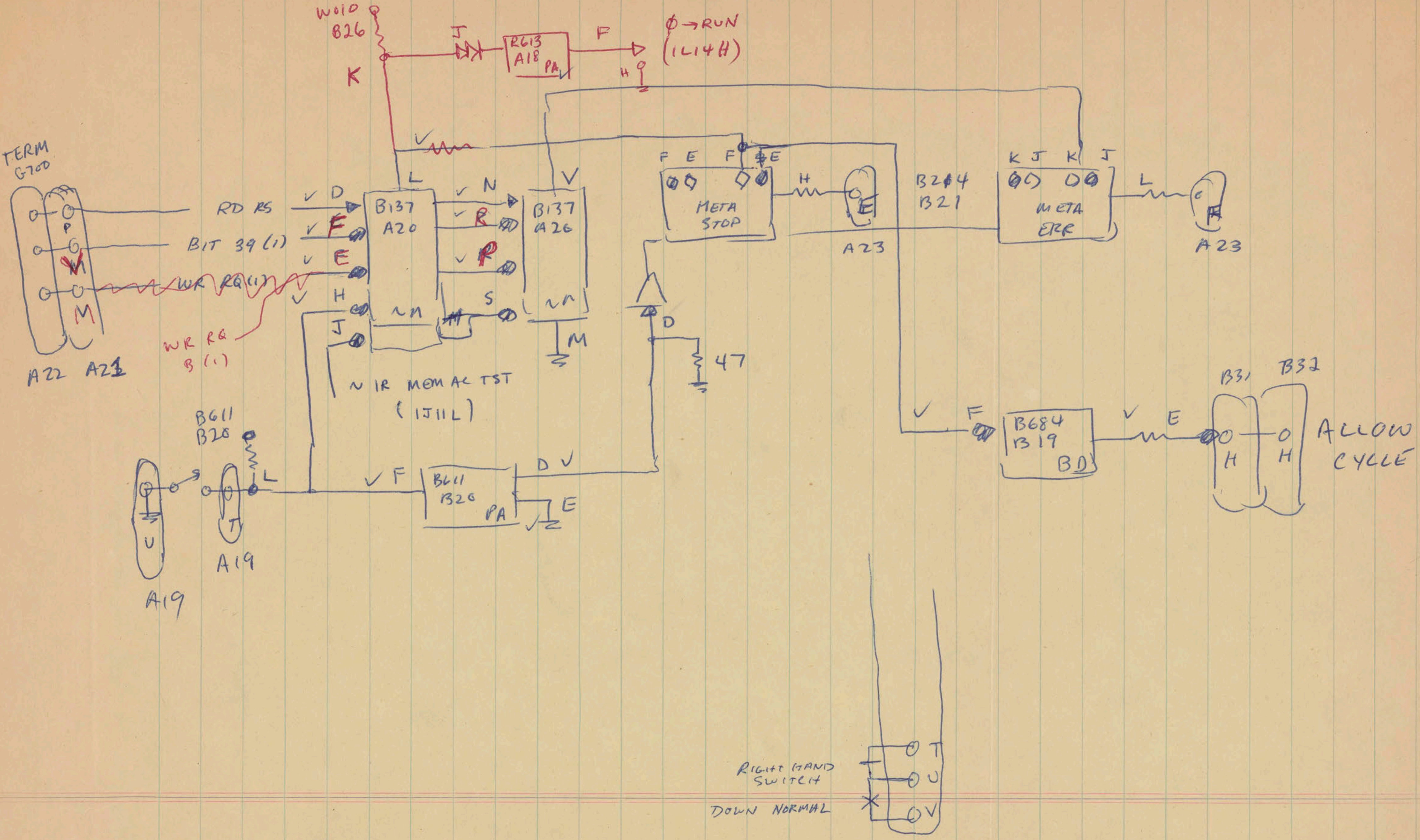
or

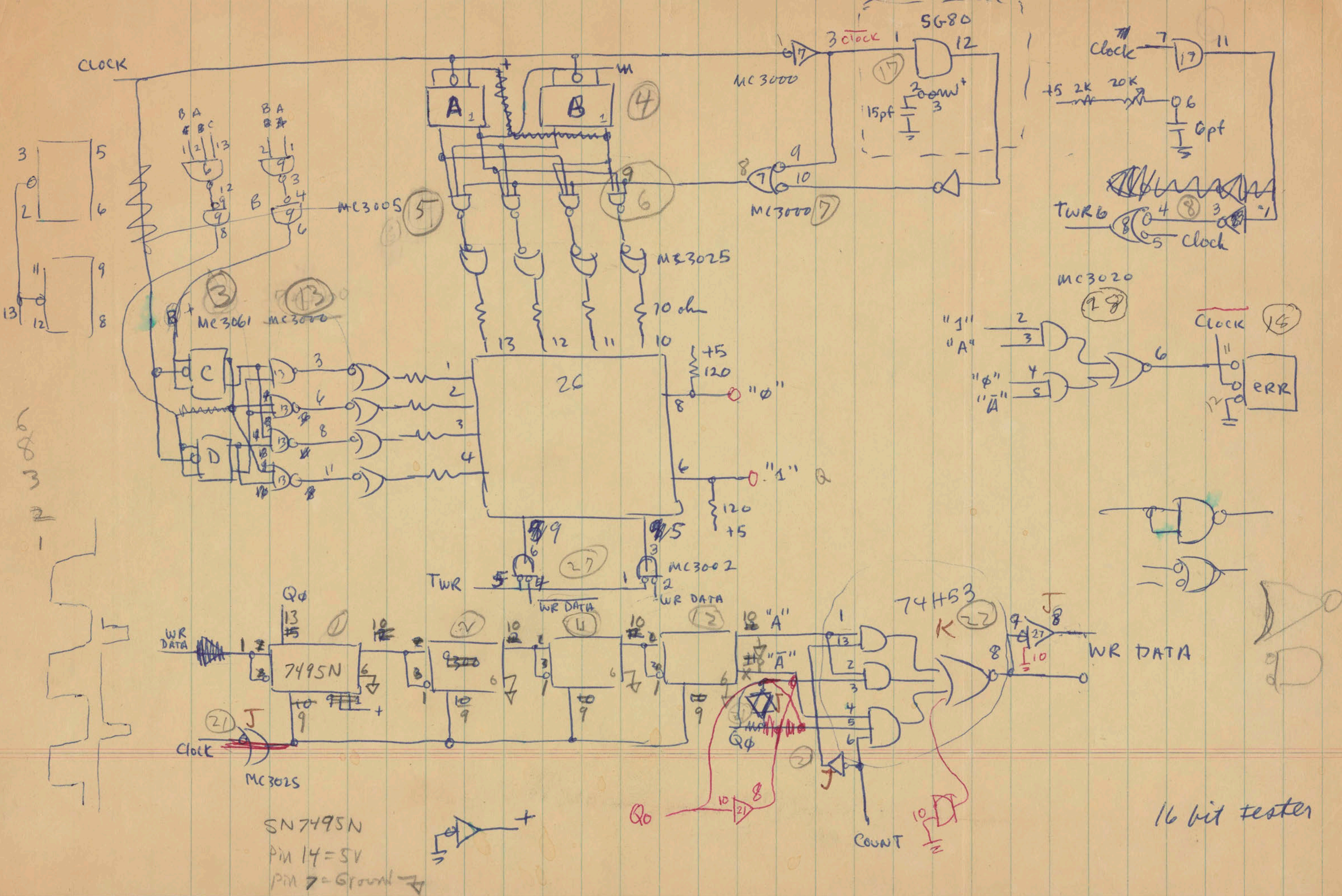




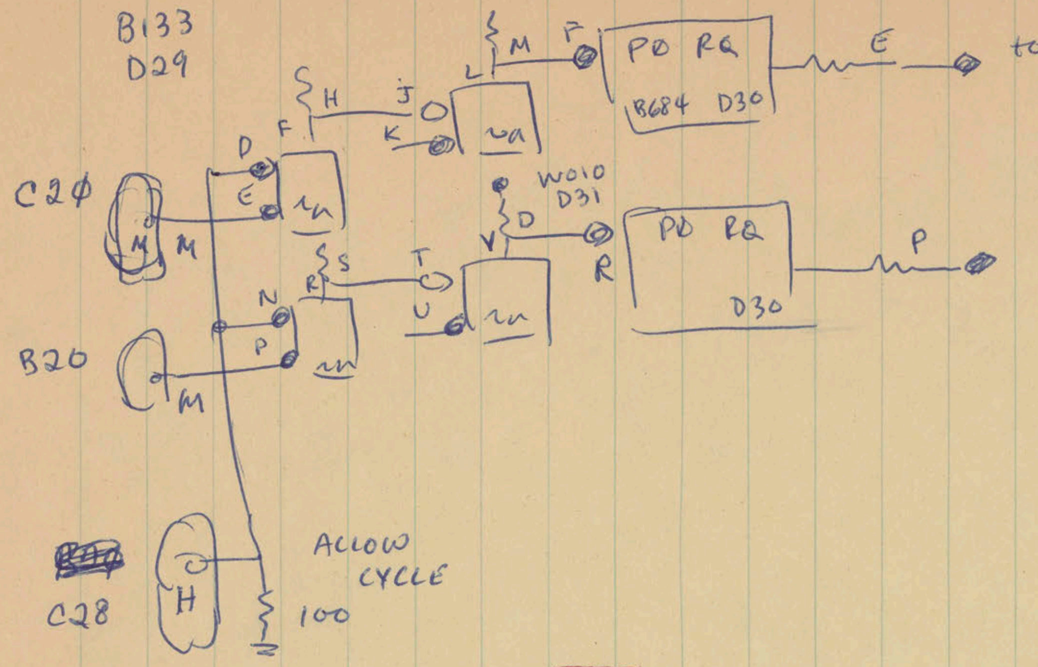


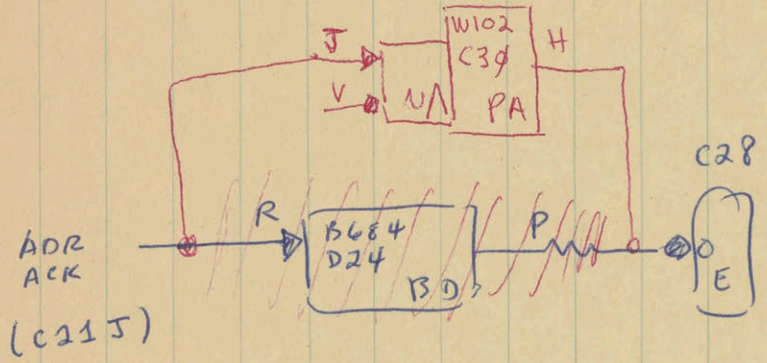
	16	17	18	19	20	21	22	23	24	
			R613	[illegible] SW	[illegible] B13	CBL	CBL	LD CBL	B156	A
			B311	B684	B611	B214	B214	B163		B

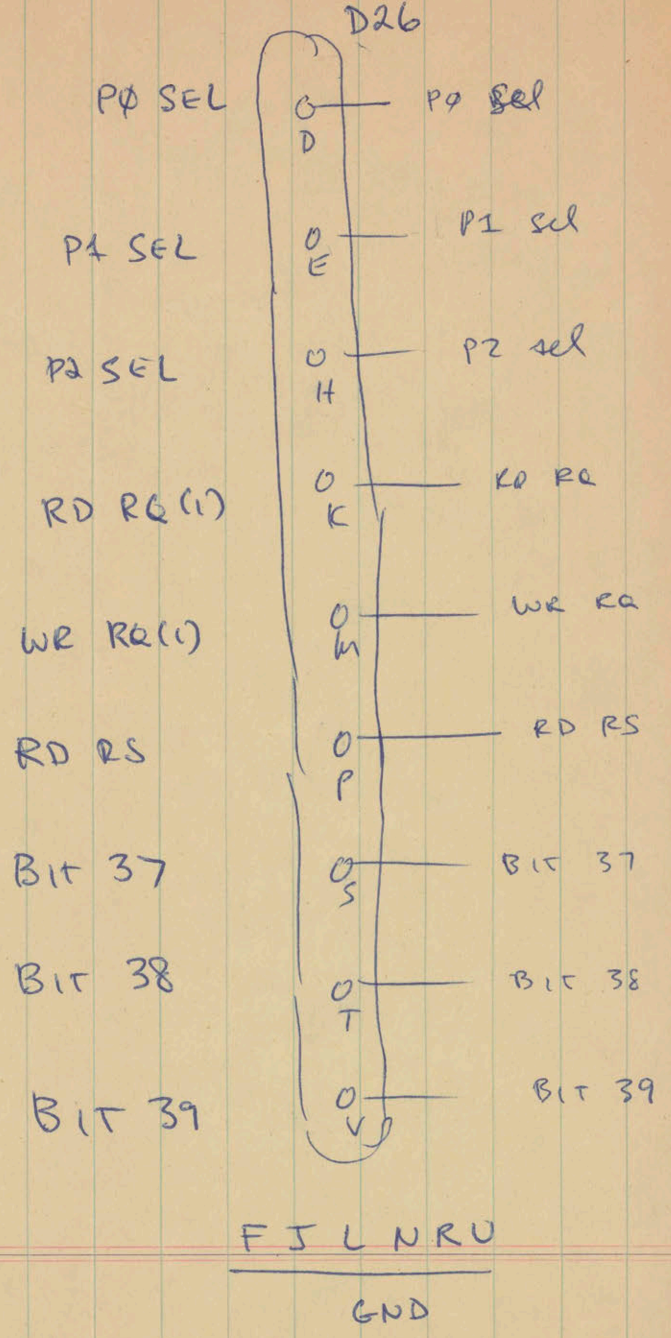

W010
B26
K
J
R613
A18
PA
F
Ø→RUN
(1L14H)
H
TERM
G700
P
W
M
M
A22 A21
RD RS
BIT 39 (1)
WR RQ (1)
WR RQ
B (1)
D
F
E
H
J
B137
A20
L
N
R
R
S
B137
A26
V
M
N IR MEM AC TST
(1J11L)
F E F ØE
META
STOP
H
E
A23
D
47
B204
B21
K J K J
META
ERR
L
A23
B611
B28
L
A19
A19
U
T
F
B611
B20
PA
D
E
F
B684
B19
BD
E
B31
B32
H
H
ALLOW
CYCLE
RIGHT HAND
SWITCH
DOWN NORMAL
T
U
V



16 bit tester

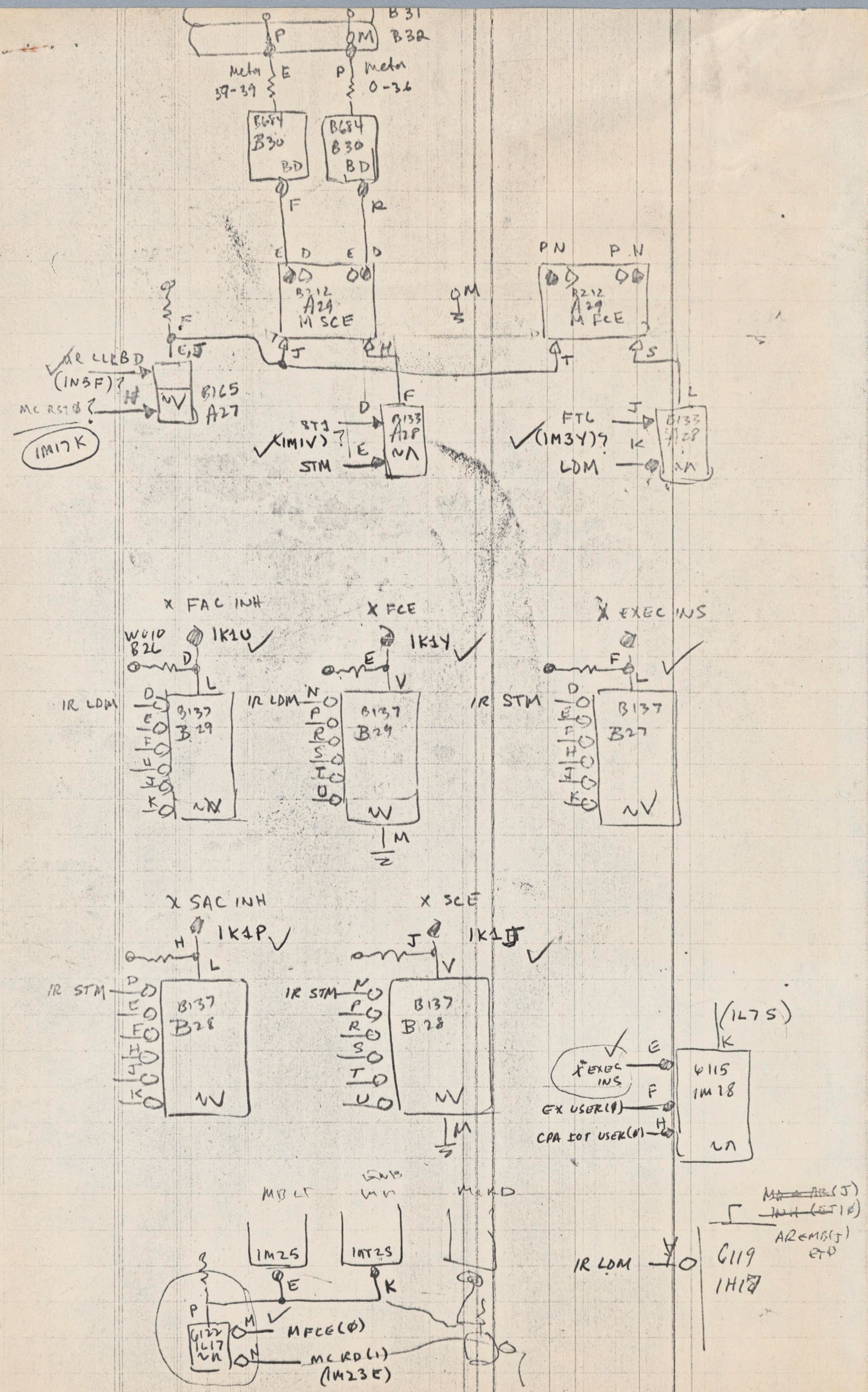






Get modules for rest of IO Bus Switch

make 100 ohm


B31
P
M
B32
Meta 37-39
E
P
Meta 0-36
B684 B30 BD
B684 B30 BD
F
R
E D E D
B212 A29 M SCE
P N P N
B212 A29 M FCE
OM
F
E,J
J
H
T
T
S
AR LLBD (IN3F)?
MC RST0?
H
NV
B165 A27
IM17K
D
F
BT1 (IM1V)?
E
STM
B133 A28 NΛ
FT6 (IM3Y)?
J
K
LDM
L
B133 A28 NΛ
X FAC INH
W010 B26
D
IK1U
L
IR LDM
D E F H J K
B137 B29
NV
X FCE
E
IK1Y
V
IR LDM
N P R S T U
B137 B29
NV
M
X EXEC INS
F
L
IR STM
D E F H J K
B137 B27
NV
X SAC INH
H
IK1P
L
IR STM
D E F H J K
B137 B28
NV
X SCE
J
IK1J
V
IR STM
N P R S T U
B137 B28
NV
M
(IL7S)
K
X EXEC INS
E
F
EX USER(0)
H
CPA IOT USER(0)
6115 1M28
NΛ
MB LT
GNB
MCRD
IM25
IMT2S
E
K
P
M
N
6122 IL17 NΛ
MFCE(0)
MCRD(1) (IM23E)
~~MA AB~~(J)
~~INH (ET10)~~
AREMB(J) ETD
IR LDM
6119 1H17

~ 2 ~

A	B150	B156	B165	B165	B133	B132		B611	W102
B			W010	B137	B137	B127	B684	TURN	CBL

24 28

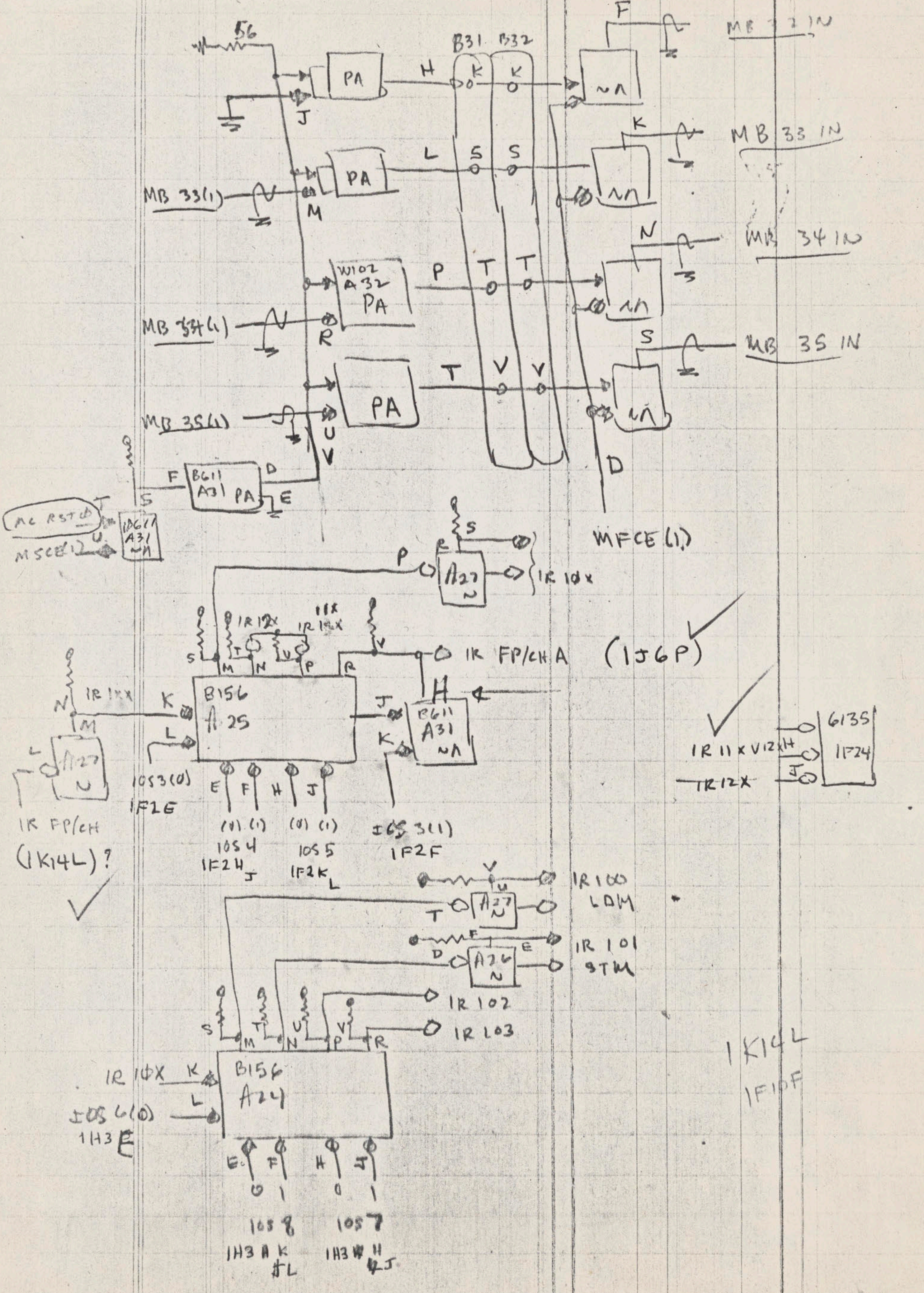
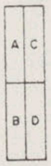


Table 7-1
Memory Bus Signals

SYSTEM MODULE PIN	PIN	MEMORY CABLE #1				MEMORY CABLE #2			
(B)	D	ADR ACK	←	MADR 22 (1)	—◆	MBD 0	↔	MBD 18	↔
(C)	E	RDRS	←	MADR 23 (1)	—◆	MBD 1	↔	MBD 19	↔
(D)	H	WRRS	→	MADR 24 (1)	—◆	MBD 2	↔	MBD 20	↔
(E)	K	PARITY	↔	MADR 25 (1)	—◆	MBD 3	↔	MBD 21	↔
(F)	M	REQ CYC	—◆	MADR 26 (1)	—◆	MBD 4	↔	MBD 22	↔
(H)	P	MADR 22 (0)	—◆	MADR 27 (1)	—◆	MBD 5	↔	MBD 23	↔
(K)	S	MADR 18 (0)	—◆	MADR 28 (1)	—◆	MBD 6	↔	MBD 24	↔
(L)	T	MADR 18 (1)	—◆	MADR 29 (1)	—◆	MBD 7	↔	MBD 25	↔
(M)	V	MADR 19 (0)	—◆	MADR 30 (1)	—◆	MBD 8	↔	MBD 26	↔
		A ↑	B ↓	C ↑	D ↓	A ↑	B ↓	C ↑	D ↓
(N)	D	MADR 19 (1)	—◆	MADR 31 (1)	—◆	MBD 9	↔	MBD 27	↔
(P)	E	MADR 20 (0)	—◆	MADR 32 (1)	—◆	MBD 10	↔	MBD 28	↔
(R)	H	MADR 20 (1)	—◆	MADR 33 (1)	—◆	MBD 11	↔	MBD 29	↔
(T)	K	MADR 21 (0)	—◆	MADR 34 (1)	—◆	MBD 12	↔	MBD 30	↔
(U)	M	MADR 21 (1)A	—◆	MADR 35 (1)B	—◆	MBD 13	↔	MBD 31	↔
(V)	P	MADR 35 (0)	—◆	RD RQ	—◆	MBD 14	↔	MBD 32	↔
(W)	S	MADR 35 (1)A	—◆	WR RQ	—◆	MBD 15	↔	MBD 33	↔
(X)	T	FMC SELECT	—◇	IGN PARITY	←	MBD 16	↔	MBD 34	↔
(Y)	V	FMC SELECT	—◆	MADR 21 (1)B	—◆	MBD 17	↔	MBD 35	↔

CABLES MUST BE LOCATED
ON THE LEFT OR RIGHT HALF
OF A MOUNTING BLOCK



CABLE SLOTS (4) DEVOTED TO EACH
CONNECTOR AS SEEN FROM WIRING SIDE

NOTE:

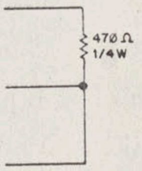
—◆ FROM PROCESSOR

◆— TO PROCESSOR

PINS:
C,F,J,L,N,R,U ARE GROUNDED

THE FMC SELECT LINES ARE
PERMANENTLY FALSE IN PDP-10
MEMORY SYSTEMS.

Table 7-2
Pin Assignments

PIN	MPX CONTROL CABLE		W990 PROCESSOR CARD USED WHEN MULTIPLEXOR NOT USED
B	−15V	①	
E	REQ N	—◆	
H	ACK N	◆—	
K	MPX CLR	→	
M	—⋀⋁⋀—●	①	

DIRECTION:
—◆ TO MULTIPLEXOR
◆— TO PROCESSOR

NOTES: PINS C,F,J,L,N,R, AND V MUST BE GROUNDED

① THE −15V CONNECTION AND THE CLAMPED LOAD AT PIN M SHOULD BE SUPPLIED AT THE PROCESSOR END TO FACILITATE OPERATION OF THE PROCESSOR WITHOUT THE MULTIPLEXOR

Kurt K

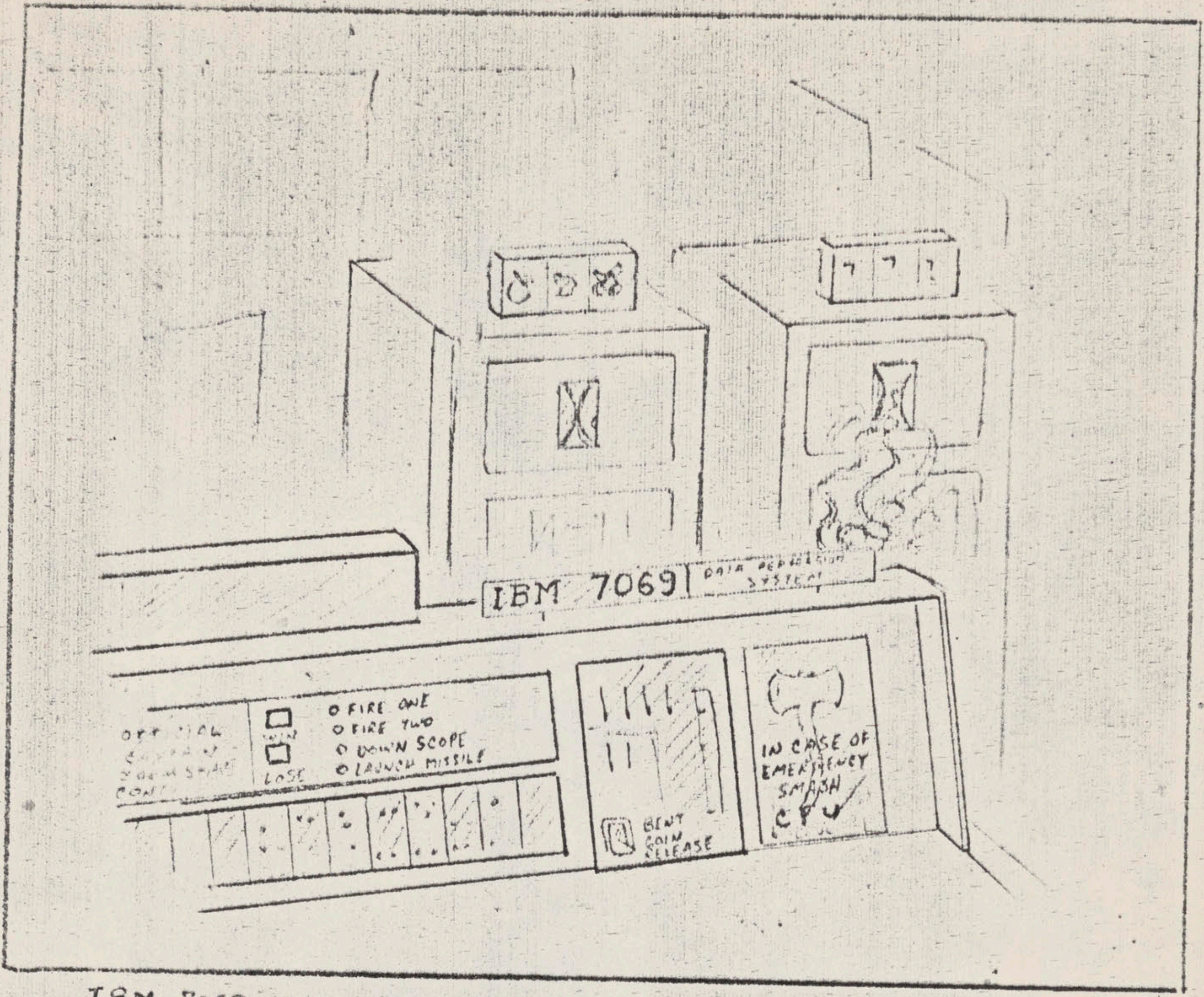
FILE NO. 7069-0
LRMWACG

IBM Systems Reduction Library

IBM 7069 Principles of Nonoperation

This manual provides no information whatsoever on the content and operation of the IBM 7069 Data Processing System. It does, however, contain detailed discussions of computer instructions, commands and orders; data channel operations, input and output equipment, and programming errors for the 7069 data Perversion system, and detailed specifications for the IBSCREW system, the STUTTER language, and the STRETCH/YAWN modifications.

This manual, together with the IBM 7069 Model II Data Perversion System bulletin, Form A69/6969, is to be used as a reference for the 7069 II system. Compatibility of like 7069/7069II instructions and operations cannot be assured, especially if these operations and instructions are employed as stated herein, nor can we assure you of anything else if the power switch is on.



IBM 7069 DATA PERVERSION SYSTEM

IBM 7069 DATA PERVERSION SYSTEM

The increased demands for precision, speed and versatility made by the new generation of computer hackers calls for a new generation of computer conceived and designed for greater hacking efficiency. The IBM 7069 DATA PERVERSION SYSTEM fills the need for an absolutely indestructable machine built to stand up to the rigors of normal use by MIT undergraduates. Unfortunately, in order to achieve this, all components and instructions which achieve any useful and meanigful operations had to be sacrificed. The 7069 is industry's answer to the rising tide of utility and efficiency which is currently overrunning the data processing field, a giant step backward in modern technology.

SYSTEM FUNDAMENTALS

The IBM 7069 Data Perversion System is absolutely incompatible with any other equipment ever designed by IBM or any other corporation. It is ridiculously slow and executes most operations with time-delay relays (SPDT). Features include:

FLOATING POINT OPERATIONS (Hot and cold running taps)
SINKING POINT OPERATIONS
FIXATED MULTIPLY
DIVIDE BY ZERO
CONDITIONAL LOSE OPERATIONS
UNCONDITIONAL SURRENDER OPERATIONS
ONE RIDICULOUSLY INCOMPETENT INDEX REGISTER "" ""
INDIRECT ADRESSES (Except where RFD or First Class mail only)

Core storage cosists of 69k half-eaten apples in plug-board racks. Word length is 12 bite or four BTB coded letters. Standard and Substandard programming assembler language is FLOP, standing for FAST*LOSE-ORIENTED PROGRAM. The actions carried out by the assembler are random and bear no resemblance to any real actions of any logical machine.

FIXED-POINT NUMBERS

When the computer encounters the instruction EFX (enter fixed-point mode), the accumulator is filled with fast-hardening epoxy cement. Upon encountering the instruction LFX (leave fixed-point mode), nothing is done, since it is obviously impossible to remove epoxy. The assembler breaks down and cries.

FLOATING-POINT NUMBERS

Entering floating point consists in filling the accumulator and IR with tap water, or alternately, Charles River Crud. This helps to lubricate the flip-flops and makes for faster operation. Overflow sensing is handled by means of a bucket which should be emptied periodically by the building janitor.

REGISTERS:

THE ACCUMULATOR: is a floating free-running conglomeration-loaded binary-tertiary augmented working register, which has room for up to three bits, or four standing up without crowding.

THE INDEX REGISTER: may be used for ineffective addressing, either in normal no-hands pigeon-holing storage, or in floating point double ambiguity mode.

THE MQ REGISTER: will not be much help, since it got so little work that it atrophied from disuse. It is usually asleep, but even when pressed into service it is ridiculously incompetent, and is best left to its own devices anyway.

REGISTER FUNCTIONS:

TAGGING: is lots of fun; one bit is chosen "it" and chases all the other bits around in ring counter mode. Game ends when bit is accidentally dropped.

MULTIPLE TAGGING: means, as near as we can figure by American League rules, that player is out at base, except on foul, where umpire calls for a "do-over".

Many other operations are possible, and we're sure at least one could be found. There is also a software routine for using the AC in cash register mode, but this has already been adequately covered by your standard IBM field service contract.

A BRIEF DESCRIPTION OF THE STANDARD OPERATING INSTRUCTION SET

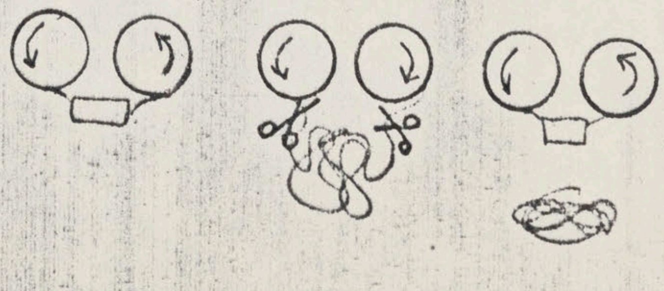
HARDWARE INTERFACE:

INSTRUCTION	FUNCTIONAL DESCRIPTION
EPC (Erase Punched Card)	--
ECS (EAT Cards)	digests object deck
PMT (Punch Magnetic tape)	--
FSM (Fold, Spindle, Mutilate)	equivalent to ECS
RWP (Rewind Printer)	very handy for typing errors
BSP (BackSpace Punch)	will also fill in accidental holes
RDR (Rotate Drum Right)	--
RDL (Rotate drum Left)	--
RRL (Drum Roll)	--
RPF (Raise Program Flag)	--
LPF (Lower Program Flag)	--
WPF (Wave Program flag)	--

LOAD/INTERCHANGE INSTRUCTIONS:

PCA (Place Complement in AC)	(no kidding)
PXA (Place Excrement in AC)	bletch!
LCN (Load Coop Number)	"Charge or cash?"
LTA (Load Teaching Assistant)	help!......
ZIP (Read Input tape into Output tape)	See 'Expensive Wire' package manual
SHFL (Shuffle and Deal)	takes AC, MQ, and XR, does two quick riffles, cut, and deal one around, including dummy.

LOGIC AND BRANCHING INSTRUCTIONS:

INSTRUCTION	FUNCTIONAL DECSRIPTION
BRI (Branch on Random Impulse)	--
BWN (Branch on Why Not)	--
BAW (Branch to Avoid Work)	--
BSO (Branch on Sleepy Operator)	--
BPO (Branch on Power Off)	--
CLB (Clear and Branch)	?
POA (Proceed On Assumption)	--
CTH (Transfer Unconditional)	Caution; this is an extremely sensitive machine, so be careful what you say.
JLN (Jump if Loud Noise)	Covered under government insecurity
STP (Halt on Red Indicator)	Rev up AC and wait
PWC (Proceed with Caution)	--
STOP (!)	Transfer directly to jail; do not pass go; do not collect $200
MRZ (Maniacal Random ZerO)	will eventually halt by zeroing itself.
XND (Exclusive And)	If not A and not B, then not.
FUR (Floating Unnormalized Randomize)	AC over easy; MQ scrambled on toast.
DER (DeRandomized)	(Universal diagnostic)
IFR (Instant Floating Retaliation)	zeros any location that dares to call and disturb it.
ADS (AC down shift)	World's only double-clutch IO
EST (Edit System Tape)	

ARITHMETIC INSTRUCTIONS:

INSTRUCTION		FUNCTIONAL DESCRIPTION
CIZ	(Clear If Zero)	--
XEM	(Exchange Exponent for Magnitude)	
SST	(Supress Trailing Significant digits)	
AAC	(Add And Clear)	--
CAC	(Clear and ADD, then Clear)	--
CHS	(Change Sign)	Replaces cute IBM 'think' sign of the week
CHSN	(Check Sign)	Endorses power bill
PAY	(***!)	Deposits an additional 35 cents for next three minutes.
FDV	(Fission Divide)	Machine splits into two 1401's, three desk calculators, and an abacus.

FILE SUPPLEMENT 7069-IBSYS-001

Standard operating systems package description

THE MONITOR: Supervisory Control of Revised Entry Work (IBSCREW):

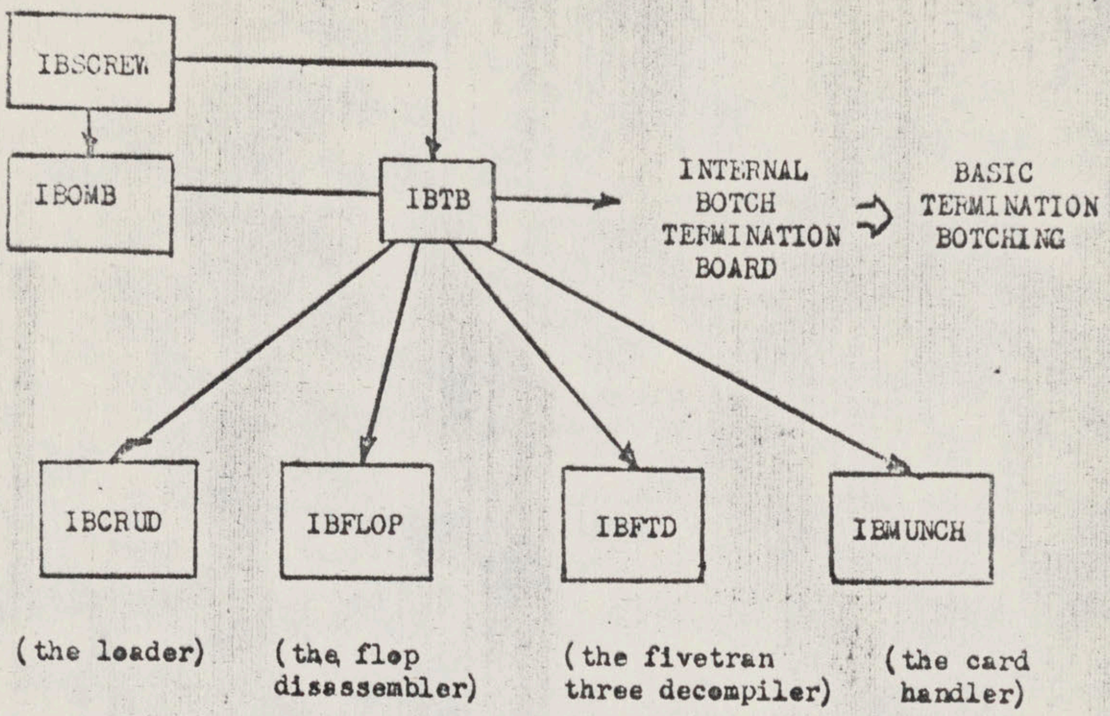
The 7069 Monitor/Operating system IBSCREW provides a variety of software options to the user, allowing him the ultimate flexitility in selecting optimal loss modes. Services provided by IBSCREW include automatic job stacking and skipping, input-output buffering for optimum data perversion, punched card and expensive wire emulation, a relocating loader with provision for unaccessed 'refugee' and missing storage of programs, and standard problem-failure oriented language.

The monitor has been designed for and around the special hardware features available on the '69, such as wired-in entropy, the automatic process procrastinator, and the patented coherence filter.

IBSCREW control cards are readily recognized by a 6-9 punch in column 1. In the card layout that follows, underlined options are those assumed by the operating system. For example, a card 6/9 IBOMB with no further options, assumes NOMACHINE, NOCORE, and NOPROGRAM.

USER (LOSER) LANGUAGE has been implemented in accordance with the standing policy of upwards, downwards and sideways compatibility; in an attempt to, as it were, go FORTRAN II½ one better, the new language has been christened FIVETRAN III.

The monitor: IBSCREW



col. 1	field	var field	options field	action:
6 9	IBSCREW	program name	[machine,*][core,*][program,*] [no machine][no core][no program]	returns control to mon.
	IBOMB		[lose, immediate - imscrew] [win,* later " " lscrew]	
6 9	IBTB	program name	[go,*] [no go]	—
6 9	IBTRY	program name	[hope,*] [(now) (never)] [no hope] [(maybe)]	—
6 9	IBCRUD	program name	[options,] [no options]	—
6 9	IBFLOP	program name	[options,] [no options]	call disassembler
6 9	IBFTD	program name	—	fivetran three decompiler

* These options are not available, now or ever.

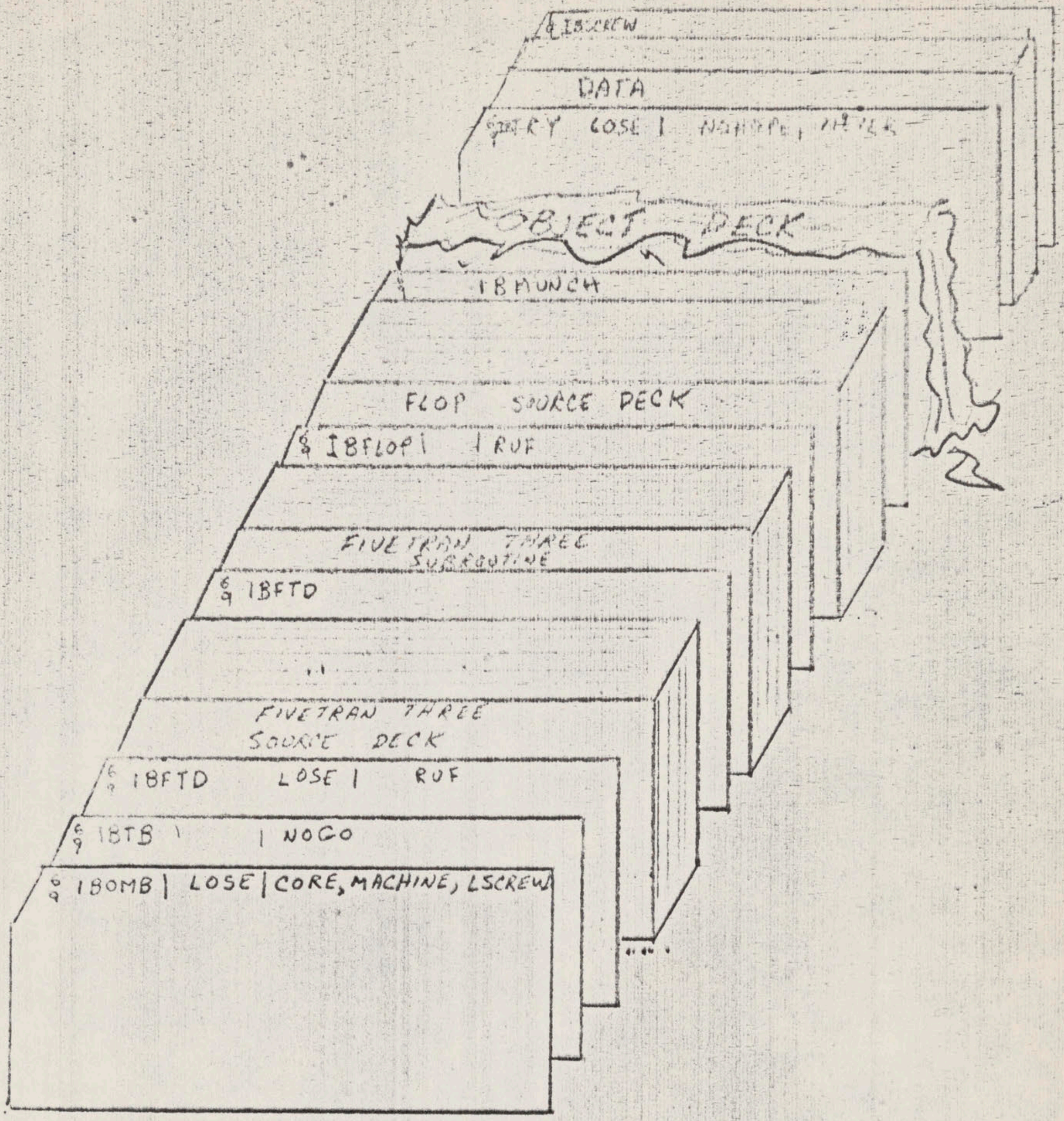


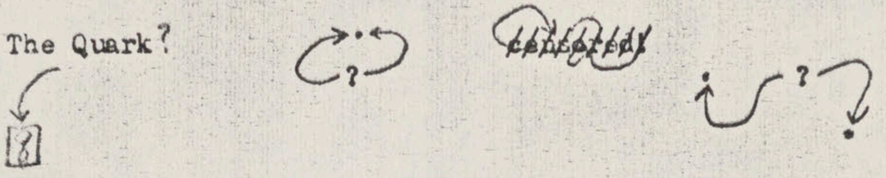
Fig. 1 - **Sample deck structure**

TECHNICAL MEMO 7069-LISP-002

A Brief Description of STUTTER as a symbolic Language

STUTTER is the symbol manipulation language of the standard 7069. The basic object is the F expression, so named by programmers who had to use it. This data structure uses bi-directional pointers. As every LISP cell contains two unidirectional pointers, every STUTTER cell contains one bidirectional pointer, thereby providing the same amount of information.

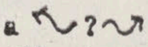
Corresponding to LISP "atoms" are STUTTER "quarks". As everyone knows, no one has yet found the quark. Some sample F expressions are diagrammed in figure 1.



The basic functions in STUTTER are:

cantr-- gets the object at which the argument points.
It is similar to the LISP cdr.

quarkp-- t iff the argument is a quark

conf-- connects two quarks with a 
Example: (conf a,b) takes the two
quarks a and b and returns

Note that the contents of a and b have
been lost.

The F expression may therefore be defined recursively (how else?) as either a quark or the conf of two quarks.

The most important innovation of STUTTER is the garbage collector, which is called when free storage is exhausted, or when there is no more free storage; this happens very often, since memory always costs something.[1] What free storage there is usually gets exhausted quickly, as stutter isa very frustrating program. High speed garbage collection is performed by clearing memory entirely, since all that memoty contains is in the long run, garbage anyway.

More (or less) information on STUTTER may be found in:

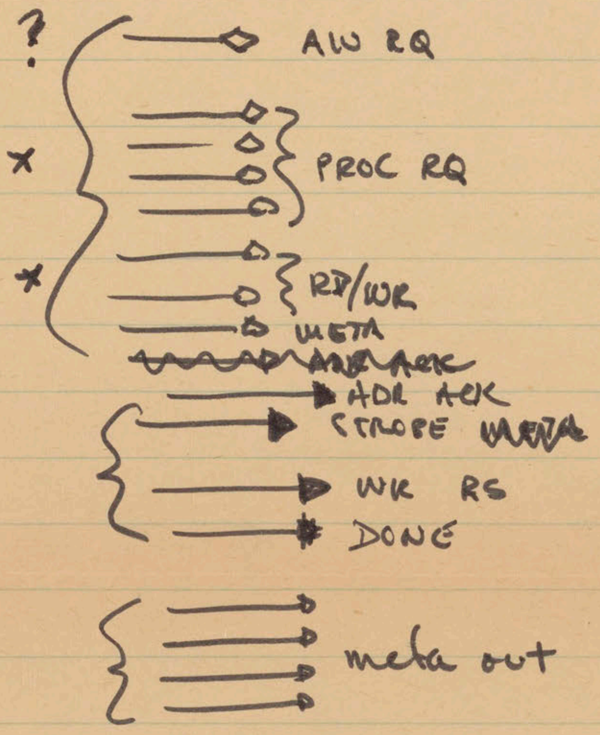
STUTTER 6.9 PROGRAMMERS MANUAL RPI Press (12 pp. $37)
STUTTER SØS: ITS USE AND ADVANTAGES Confusion Centintal (10 pp.)
STUTTER AND OTHER LOSES vol.1, no.69 Proceedings of the Cambridge Royal
Verbal Fireworks Society, M. Wand et al.

[1] OF course, the best things in life are free, which says something about the desirability of computers.

DEC INTERFACE

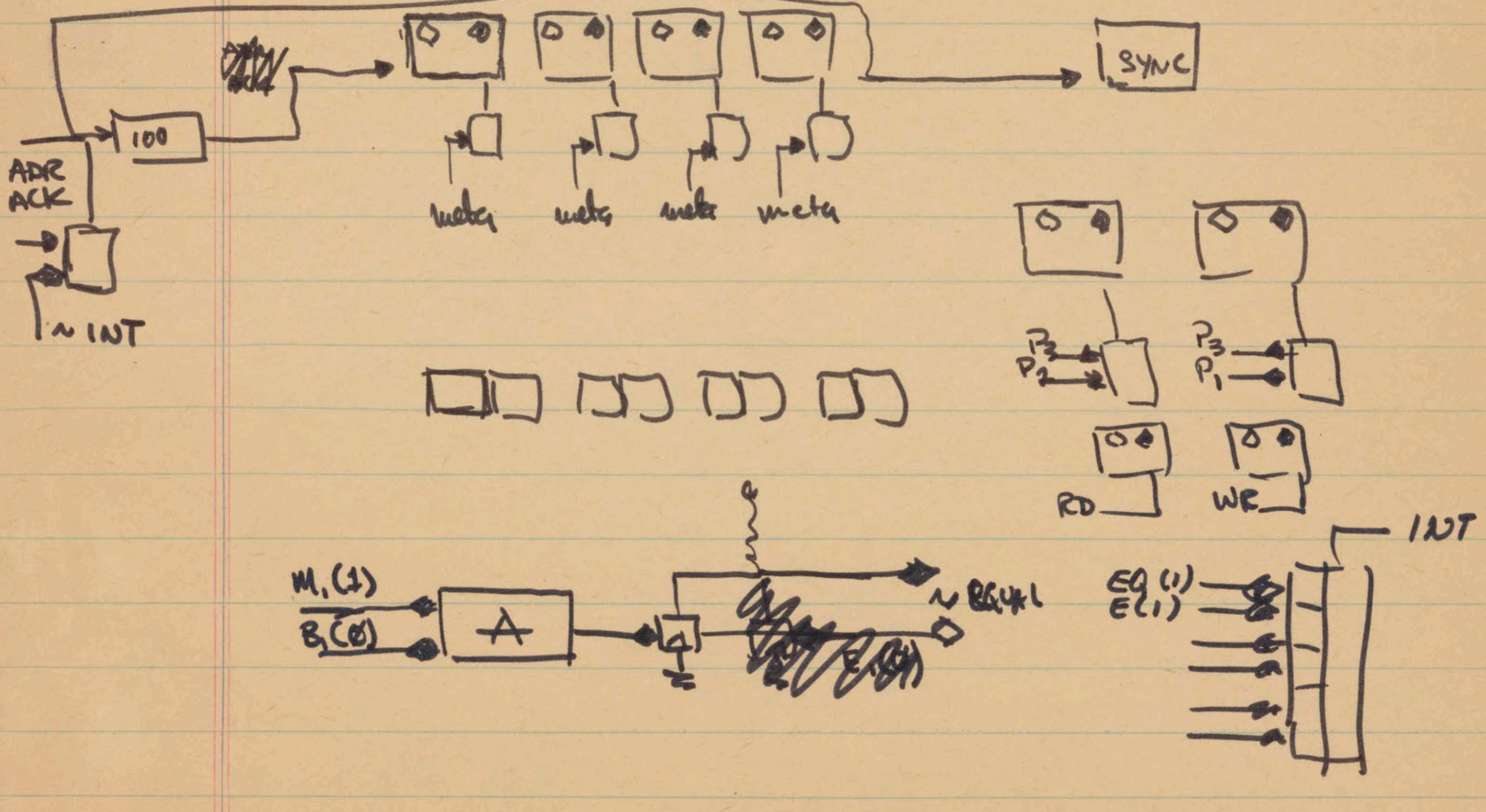
MEMORY MONITOR

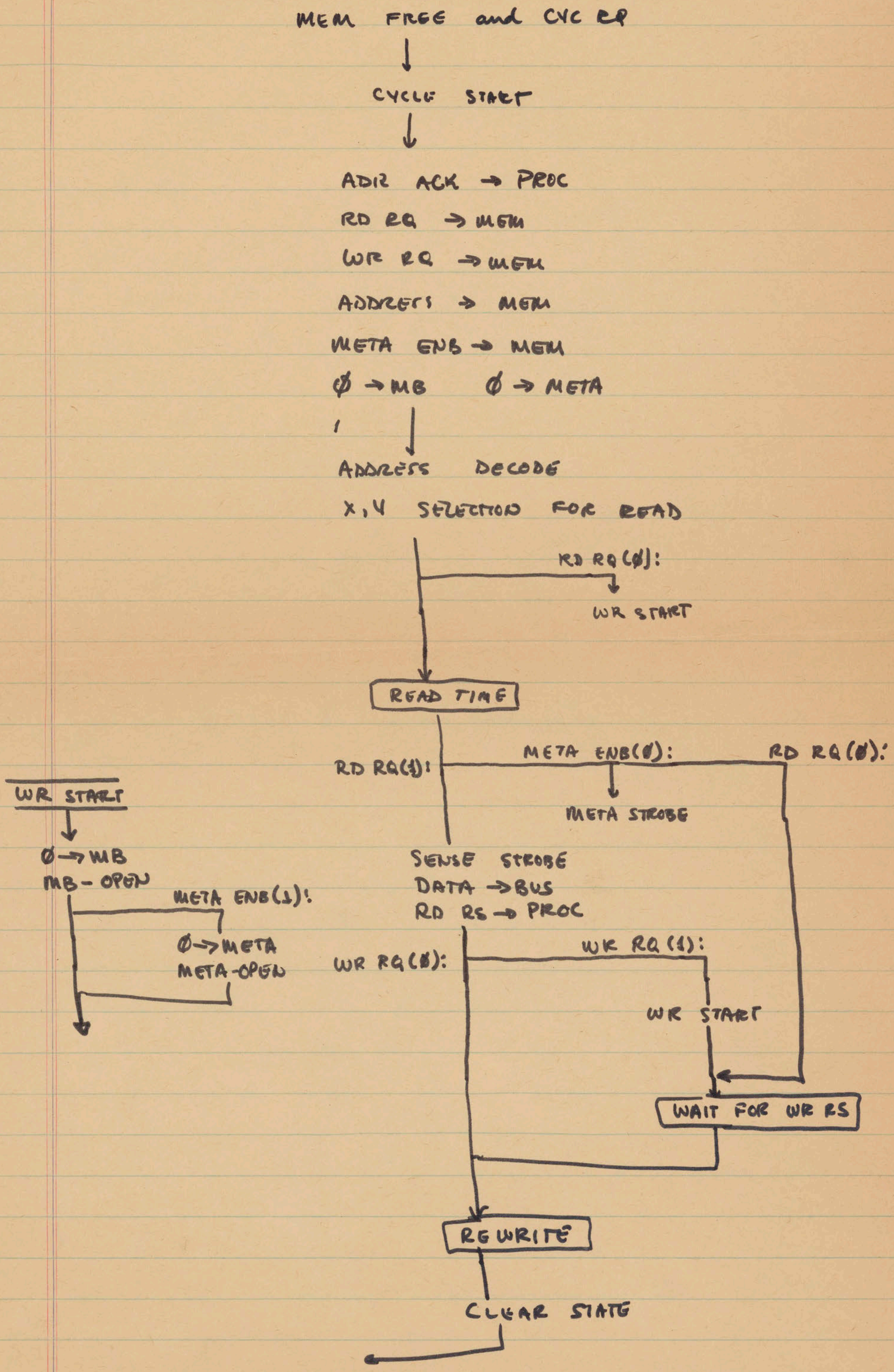
1. Need for more flexibility in control of meta bits
2. No room in fabritek monster
3. Change with differing requirements



FOR INSTANCE

1. META BIT protection on processor




MEM FREE and CYC RQ
CYCLE START
ADR ACK → PROC
RD RQ → MEM
WR RQ → MEM
ADDRESS → MEM
META ENB → MEM
Ø → MB
Ø → META
ADDRESS DECODE
X,Y SELECTION FOR READ
RD RQ(Ø):
WR START
READ TIME
RD RQ(1):
META ENB(Ø):
RD RQ(Ø):
META STROBE
SENSE STROBE
DATA → BUS
RD RS → PROC
WR RQ(Ø):
WR RQ(1):
WR START
WAIT FOR WR RS
REWRITE
CLEAR STATE
WR START
Ø → MB
MB - OPEN
META ENB(1):
Ø → META
META-OPEN

META BITS

The Project MAC PDP-6 will ,soon recieve a 262K core memory from FTI. The word ~~length~~ will be 40 bits long, wheras the standard -6 word length is 36 bits plus parity. This memo will describe changes to the -6 processor that are being considered to allow these 3 Meta Bits to become available to the programmer.

To clarify the need for processor modifications, let me describe the memory interface as it presently exists.

MEMORY BUS SYSTEM

The PDP-6 utilizes a parallel bus system of memory interconnention. The memory bus originates from the processor and carries the following signals;

36 Data bits (two-way pulse transmission)

1 Parity bit (trated as a 37th data bit)

18 address lines

1 Fast Memory Select

3 timing signals

3 cycle descriptor lines

Each unit of the memory is attached to the bus, and a special interface monitors 4 of the address signals (high order), ~~and~~ the Fast Memory Select, and the cycle descriptor signifying a processor cycle request. It determines the presence of a request for that memory unit on the bus, and initiates a cycle. It is assumed that this interface is internally connected in such a way as to assure that each unit on the bus responds to a ~~unit~~ UNIQUE code of the four address and one FM select line.

The fast memory select exists for a good reason. Normally, each mem~~ber~~ory unit responds to a fixed bit pattern in the top four address lines. This means that each unit will consist of two **14 or 16K words, and the map from the address space into physical memory locations would look like Figure 1.

META BITS

Project Mac PDP-6 is soon recieving a 262K word core memory from FTI. The word ~~length~~ will be 40 bits long, Wheras the standard -6 word length is 36 bits plus parity. This memo will describe changes to the -6 processor that are being considered to allow these 3 META bits to become available to the programmer.

To clarify the raison d'etre for the processor modifications, let me describe the memory interface as it exists now.

MEMORY BUS System

The PDP-6 utilizes a parallel bus system of memory interconnection. The memory bus originates from the processor and carries the following signals;

36 DATA BITS (two-way pulse transmiss on coax cables.)

1 Parity bit (treated as 37th data bit)

18 ADDRESS Signals

1 Fast Memory Select

3 Timing Signals

3 cycle descriptor lines.

EACH UNIT of memory is attached onto the bus, and a special interface monitors 4 of the ADDRESS signals (high order), the FAST Memory Select Signal, and the cycle descriptor signifying a processor cycle request, to determine the presence of a request for that memory unit on the bus, It is assumed that this interface is internally connected in such a way as to assure that each unit on the bus responds to a unique code of the 4 address and one FM select line.

The Fast Memory Select exists for a good reason. Normally, each memory unit responds to a fixed bit pattern in the top 4 address lines. This means that each unit will consist of 2^{14} or 16K words, and the map from the address space into physical memory locations would look like Fig 1. But the design of the processor is such that on most machines the address space looks like Fig 2., where the low 16 registers of memory are Flip-Flop Fast Memory, which are used accumulators, ~~and~~ index registers, and normal memory locations. When signaling for a request, the processor recognizes that the address it wants lies within the fast memory, and it activiates the FM Select select line. This results in ~~that~~ memory unit #1 ~~will~~ not begining a cycle, but instead the Fast memory ~~will~~ acknowledges this reference. This addressing scheme does not imply that the low ~~other~~ 16 registers of memory unit #1 are nonexistent, but rather that they are not normally referenced. In Theory unit #1 is identical to unit #2, and should The processor choose to never activate the FM select line, the address map would be as in Fig #1.

Mass memory

The mass memory will have an interface very closely resembling That of The standard 16K DEC memories, complicat'ed by The longer word length and the larger number of addressable locations.

Normally, the mass memory will recognize any request in which the top four address lines are non-zero (0001 → 1111). This will enable the existant 16K DEC memory to be used in conjunction with the mass memory. However, there is an additional signal introduced on The memory bus (the Special addressing mode), which will be used when The processor wishes to signal a reference into the low 16K of The mass memory. This will result in the address → physical memory map shown in Fig #3, Normally 0-15 are in FM memory, 16-16K are in DEC memory, 16K → 262K are in mass memory. The addresses where FM cell is not used by The processor (resultin in 0-15 in DEC 16K memory) are called <u>Shadow memory</u>, and When Special mode is used (resultin in 16→16K in Mass memory) are called Hyper memory.

TO accomadate the ~~longer~~ PDP-6 word size, The mass memory word is logically divided into two parts. one part is 37 bits long (36 + 1 parity), The other 3 bits long. Two more cycle descriptors are introduced, one assigned to each part of The word. Essentially These bits are used to reconfigure the logical word, so That it may look like 3 bits, 37 bits, or 40 bits. This is necessary because one of the The type of cycles The processor can initiate is a <u>write</u> cycle, where The current contents of the memory is read-to-zero, and The data from The processor written in its place, In This case it may be desirable that the previous contents of the most bits should not be destroyed, so it the descriptor assigned to the 3 most bits is not active, The memory will rewrite The data in that part of The word. (Fig A4).

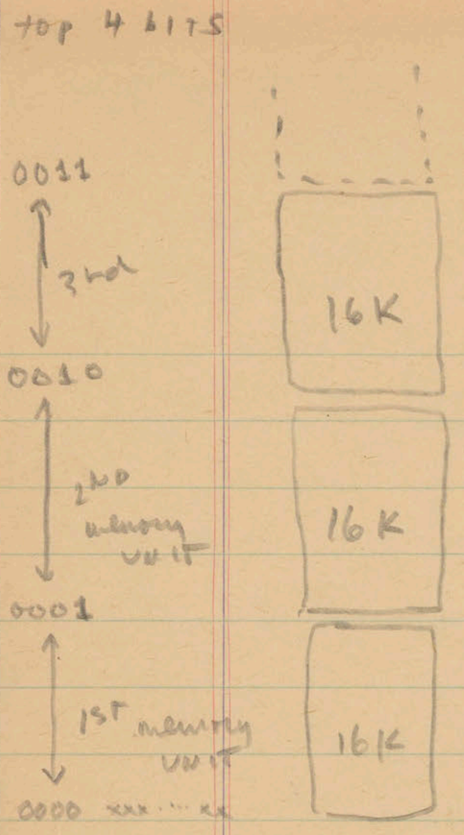


Fig 1

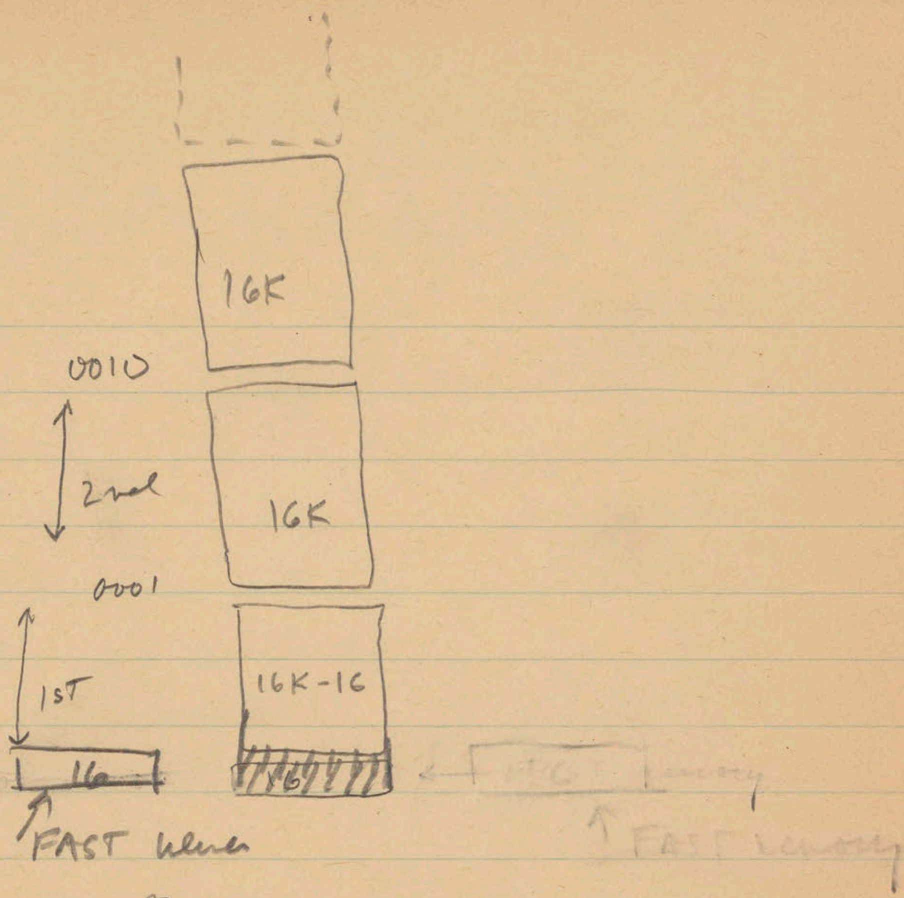
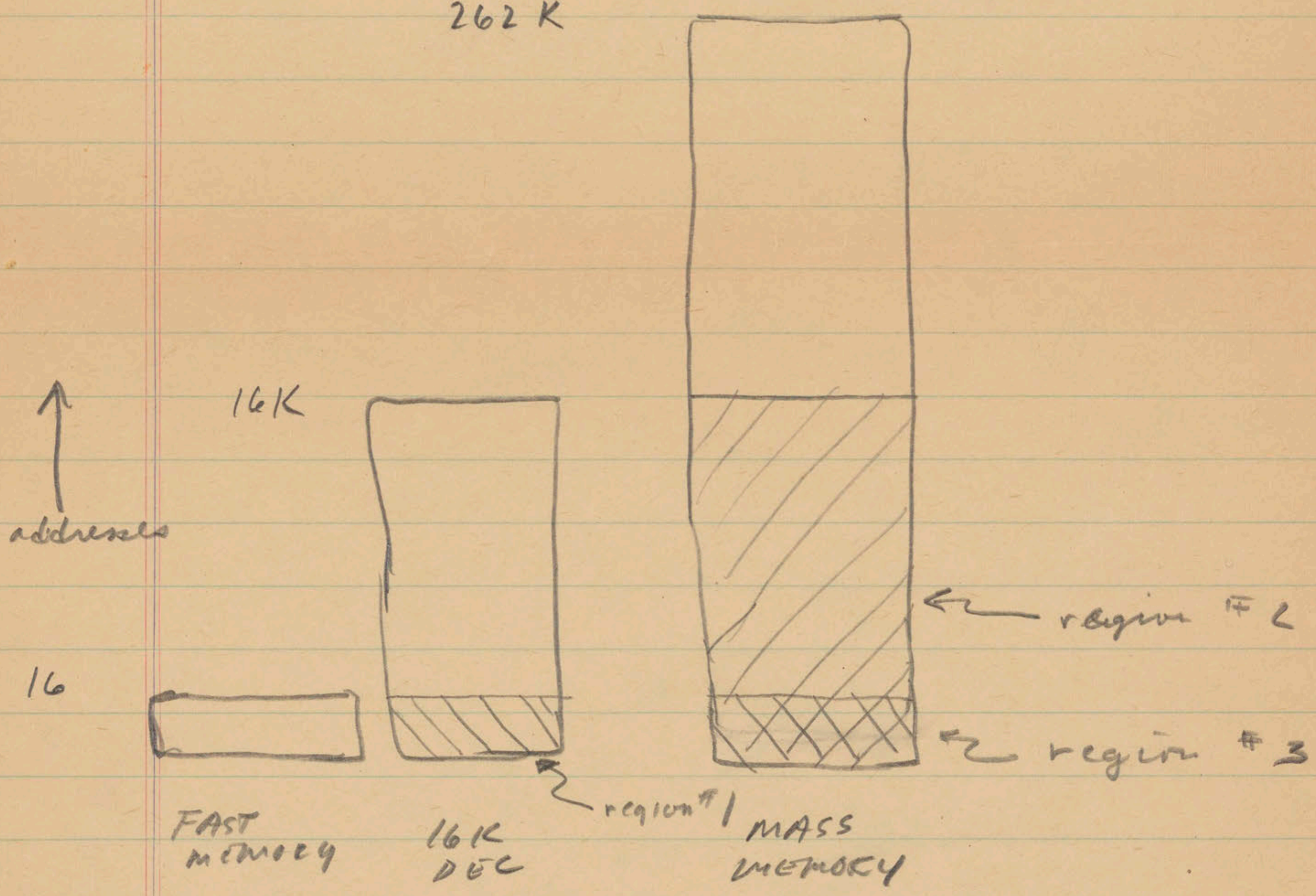


Fig 2



Normal references go to white areas

Not FM select goes to region #1

Special mode goes to region #2

Not FM select ∧ Special mode goes to region #3

Fig 3,

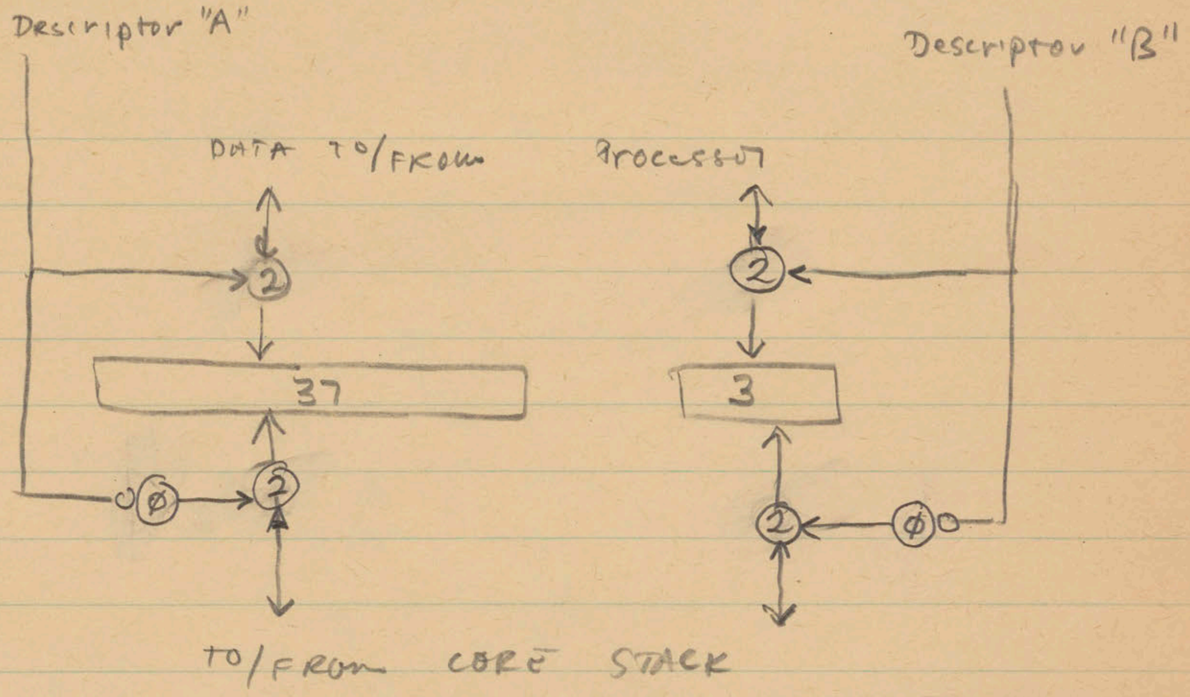
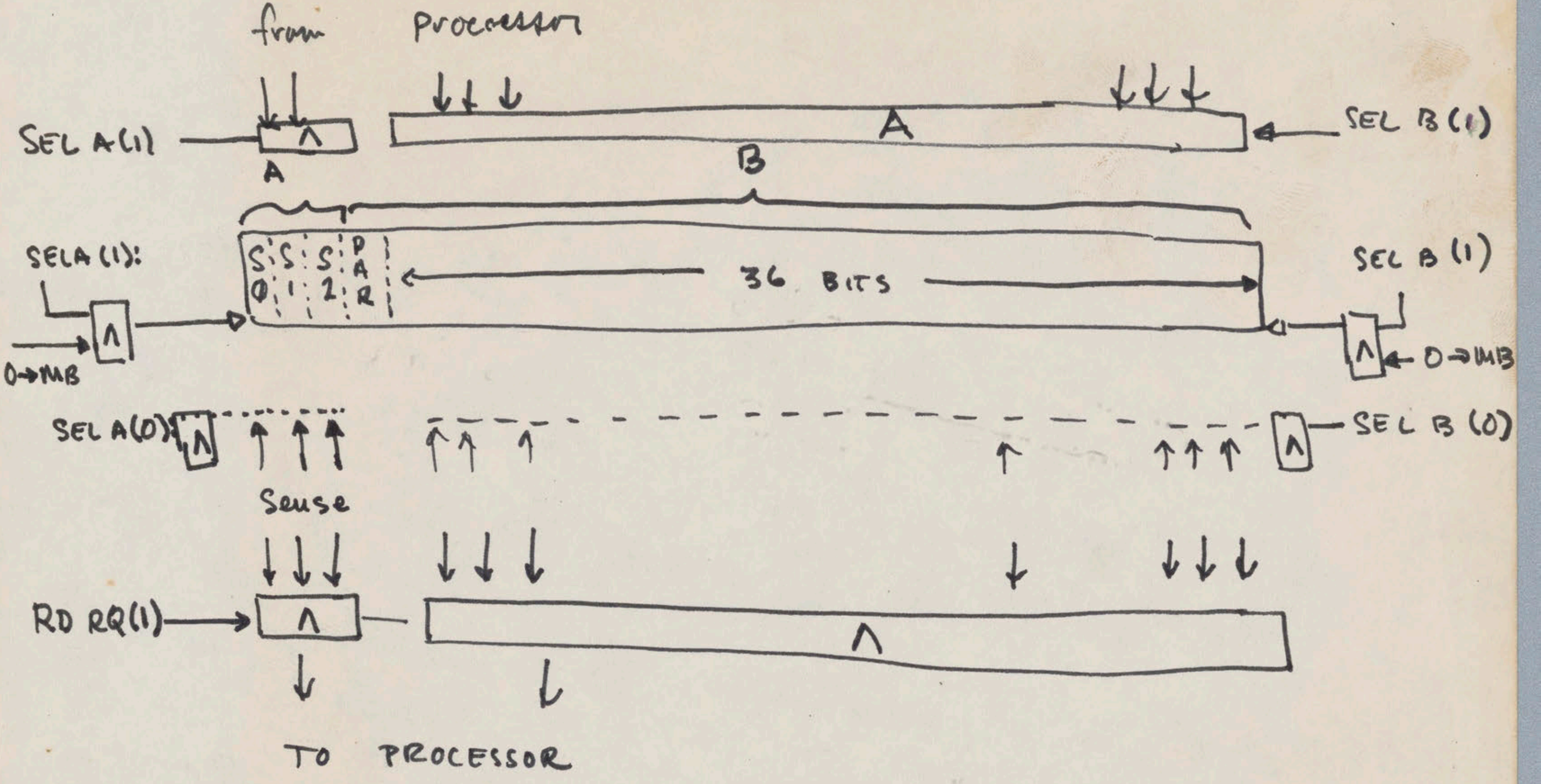

Descriptor "A"
Descriptov "B"
DATA TO/FROM PROCESSOR
2
2
37
3
∅
2
2
∅
TO/FROM CORE STACK

Fig #4


from processor
SEL A(1)
Λ
A
SEL B(1)
A
B
SELA (1):
S 0
S 1
S 2
P A R
36 BITS
SEL B (1)
Λ
0→MB
Λ
0→MB
SEL A(0)
Λ
SEL B (0)
Λ
sense
RD RQ(1)
Λ
Λ
TO PROCESSOR

Solutions for mass memory

1. HAVE META ENB PER PROCESSOR THAT:

WHEN ZERO (0)

a) causes clear of meta buffer at 2nd half of read/write to be suppressed

b) causes memory strobe to be enabled When READ(1) or meta strobe (0)

WHEN A "ONE" (1):

a) causes BUS → meta bits to be enabled (along with WRITE SYNC)

2. HAVE META STROBE APPEAR AT INTERFACE WHENEVER READ HAPPENS Along with clear/set

* 3. HAVE MEMORY STATE SIGNALS AVAILABLE AT INTERFACE

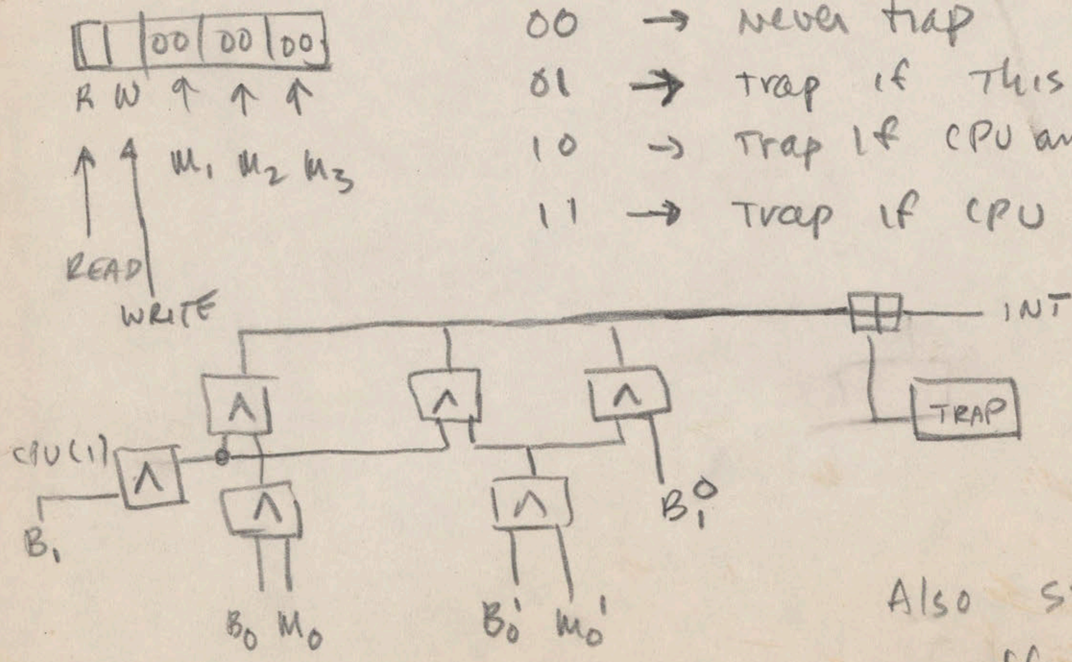
4. meta trap four states per bit

00 → Never trap

01 → trap if This bit (1)

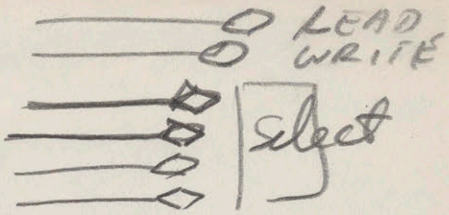
10 → Trap if CPU and bit (0)

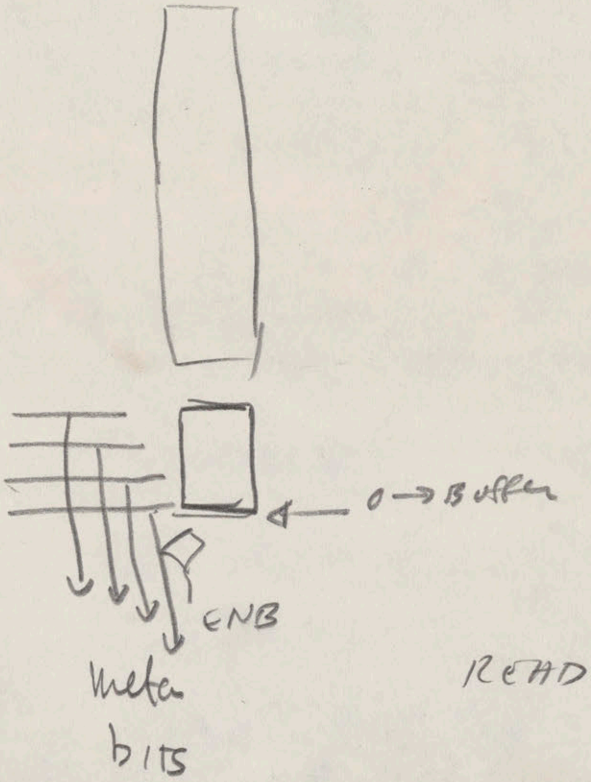
11 → Trap if CPU and bit (1)

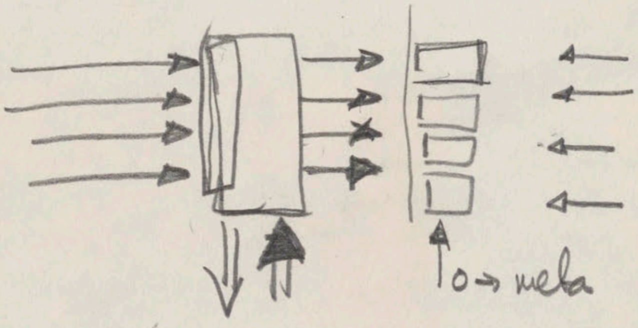


Also stores read/write

polf. pilf







Alida

READ meta → bus

TRAP SYNC

R/W meta → BUS

TRAP SYNC

↓

ADR ACK

WR RS

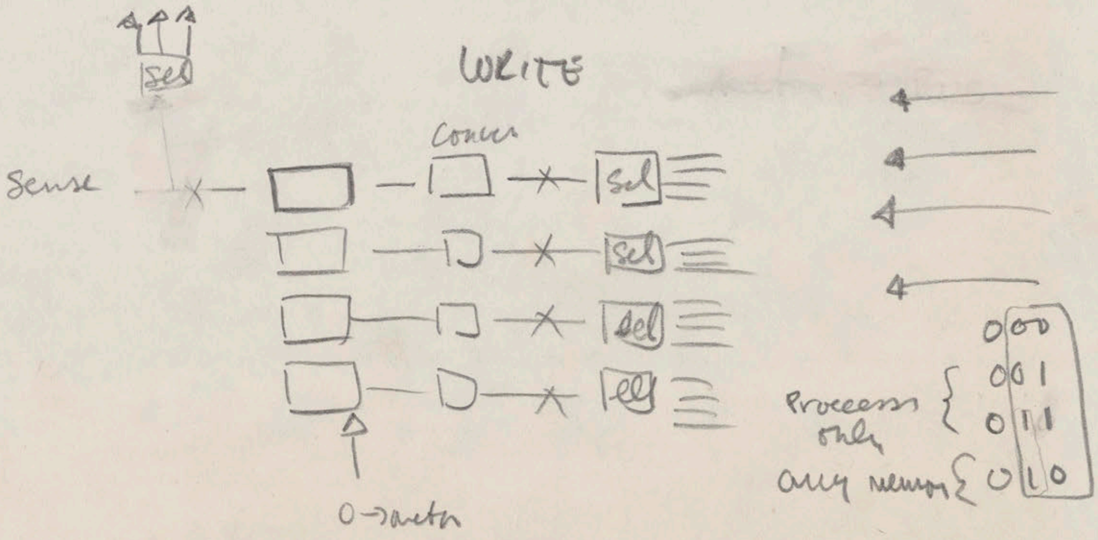
meta

0 → meta bits

1 → meta bus enable

ADR ACKV

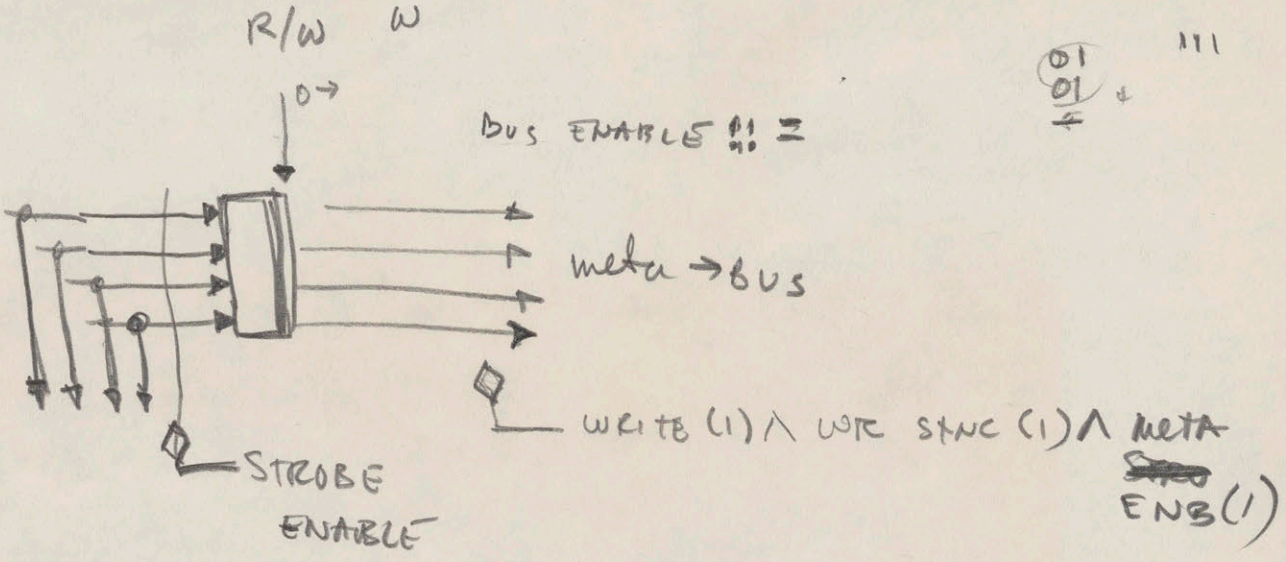
(RD RS ∧ WRITE(1))



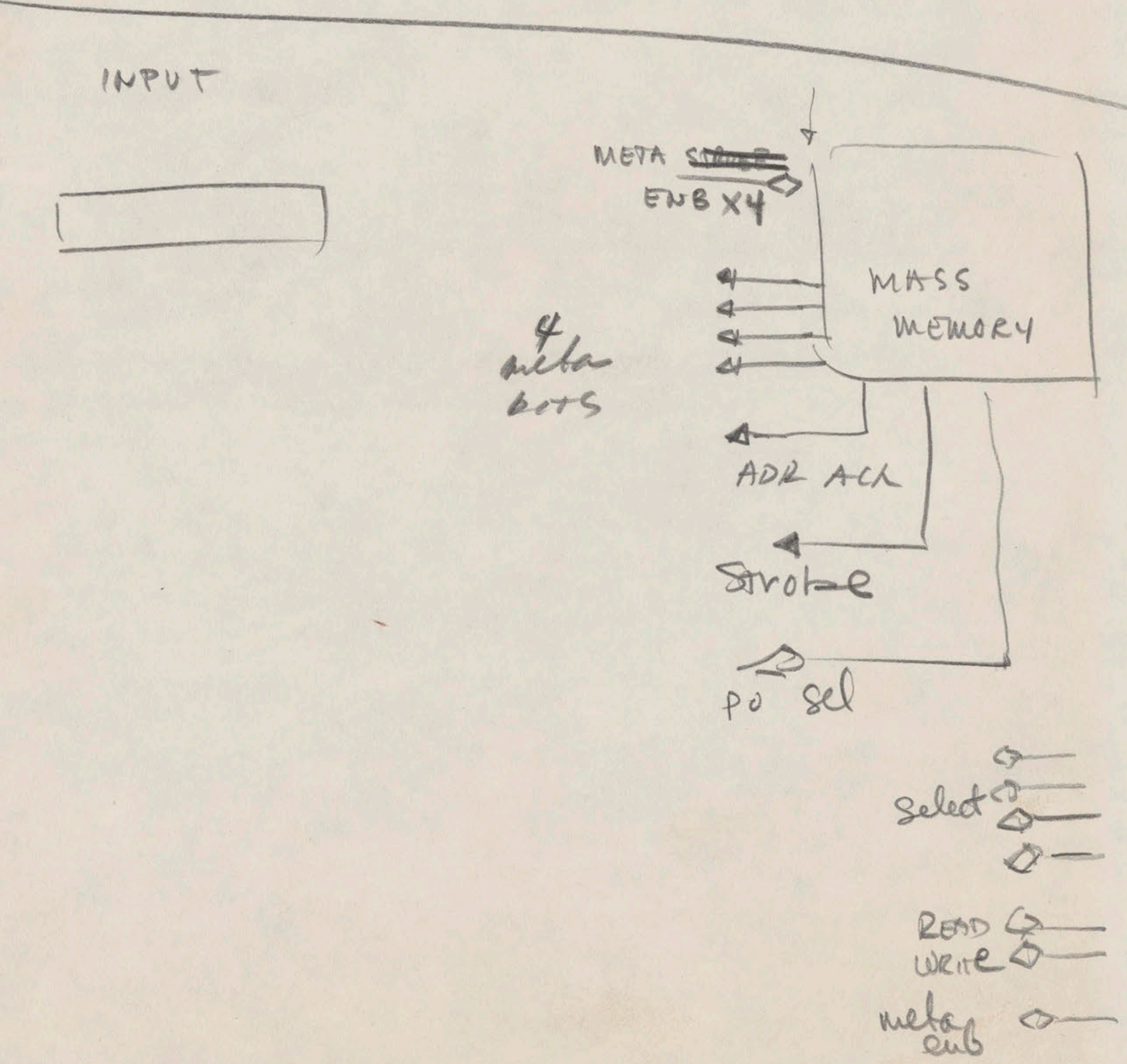
	000	do nothing
Process only	001	if bit(0)
	011	if bit(1)
any memory	010	if bit(1)

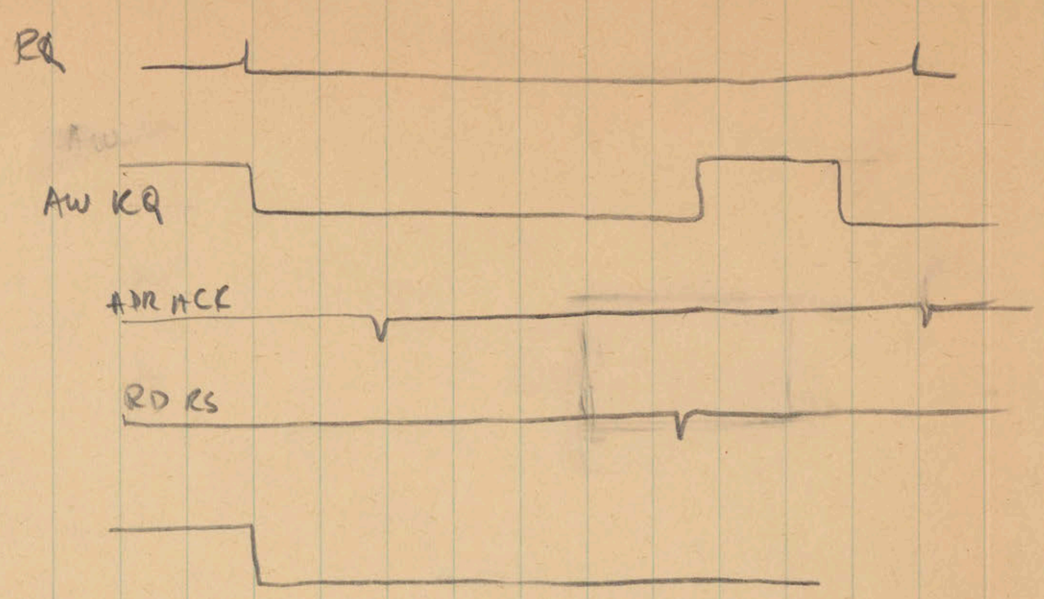
111

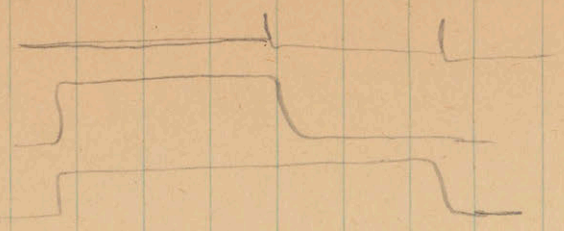
01
01
±



STROBE E ! READ (1) V (WRITE (1) ∧ META (0))

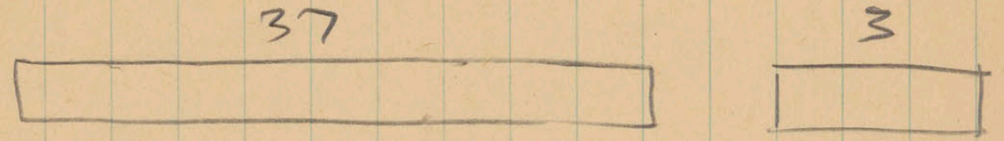






WR (Ø) : ~~R~~ RST1

WR (1) ∧ RD (Ø) : RST1



MB ← (Ø) at cycle	MB ← (Ø)
RH(1) MB ← (Ø) at R/W	LH(1) MB ← (0)
RH(Ø) or RD(1) MB ← sense	LH(Ø) or RD(1)
RH(1): MB ← MBOS	LH (1): MB

Signals

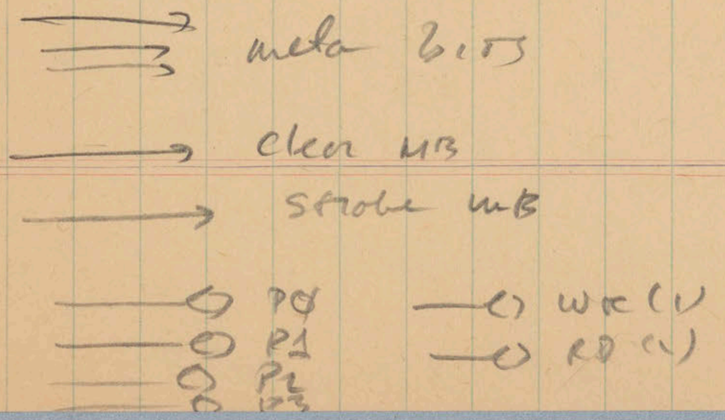
MB ← (Ø) at cycle start

MB ← (Ø) at R/W

MB ← sense if RD(1)

MB ← MBOS if WR(1)

also



~~FOR META INSTRUCTIONS~~

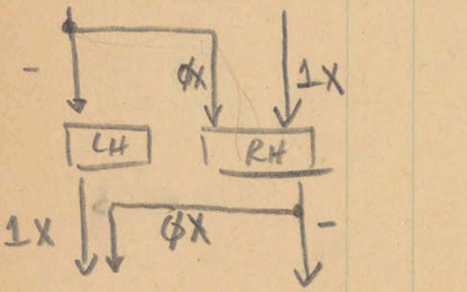
LOAD AC, ADR FC(E) META Ø1
STORE AC, ADR SC(E) META Ø1

TBNE
TBZE
TBLE
TBOE
TBNN
TBZN
TBCN
TBON
TBN
TBZ
? TBL
? TBO
4 LOAD
5 STORE

100 TBNE MSK, WORD
~~TBNN~~
~~TBZE~~
~~TBZN~~
~~TBCE~~
~~TBCN~~
~~TBOE~~
107 ~~TBON~~

110 TBZ MSK, WORD
TBO WSK, WRD

CONFIGURATION



1X ⇒ WRITE RH
ØX ⇒ DO NOT WRITE RH

X1 ⇒ WRITE LH
XØ ⇒ DO NOT WRITE LH

META (ØØ) not used

$RH_{MC} \rightarrow$

$LH_{MC} \rightarrow RH_{MB} \rightarrow LH_{MB}$

During fetch cycle for TB

AC FlD ⇒ AR

Durin ETØ for TB

AR(i) → MB(i)

[Cause gates for ACBM TO be grounded]

For TB instruction FC(E) PSE + META(Ø1)

MEMORY CONFIGURATION

META(10) normal mode

$RH_{MC} \leftrightarrow RH_{MB}$

$LH_{MC} \rightarrow LH_{MB}$

META (Ø1) For TB instructions

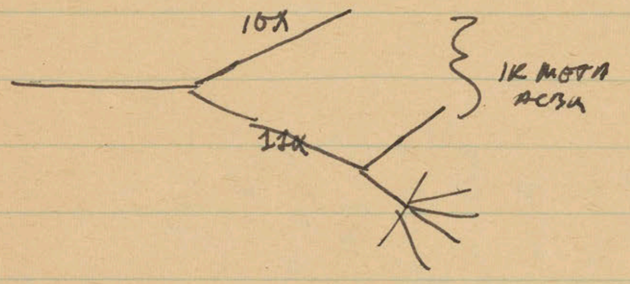
$RH_{MC} \rightarrow$

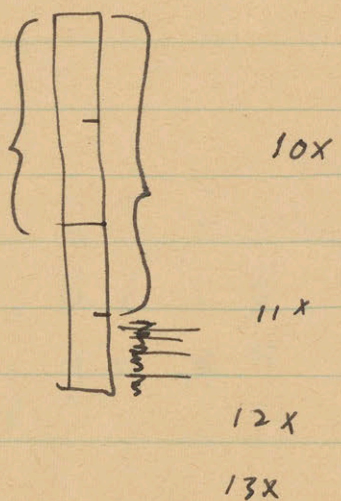
$LH_{MC} \leftrightarrow RH_{MB} \rightarrow LH_{MB}$

META (11) For MARK instruction

$RH_{MC} \leftrightarrow RH_{MB}$

$LH_{MC} \leftrightarrow LH_{MB}$



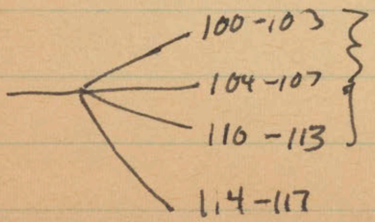


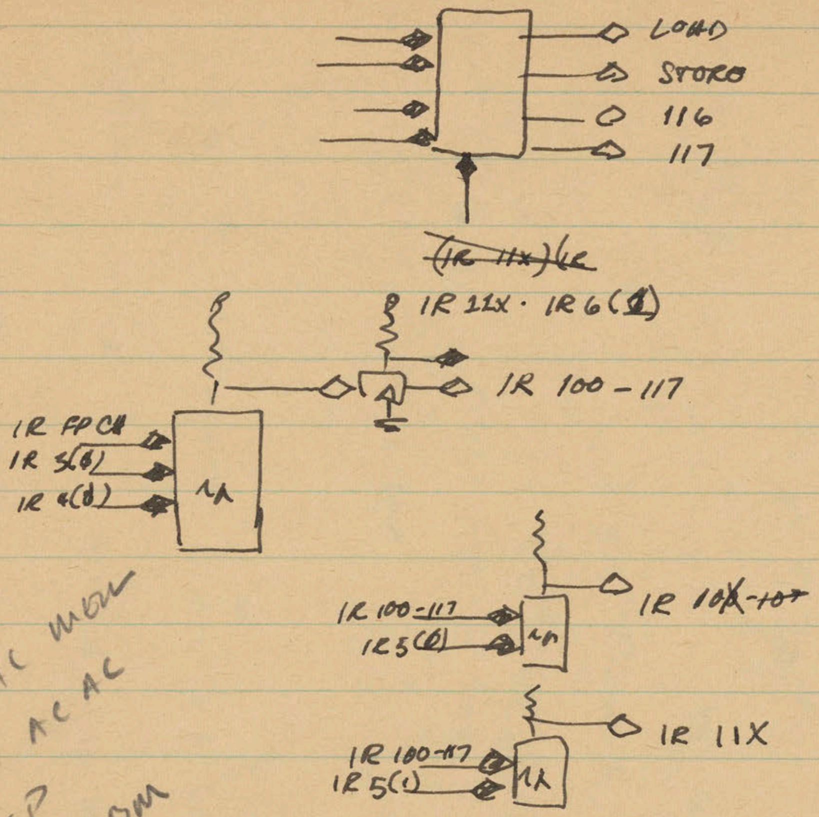
$\overline{001}$ 000 $\overline{000}$

001 001 011

IR FP/CH ∧ IR3(Ø) ∧ IR4(Ø) ∧ (IR5(Ø) ∨ IR6(Ø))

IR 10X ∨ (IR 11X ∧ IR6(Ø))





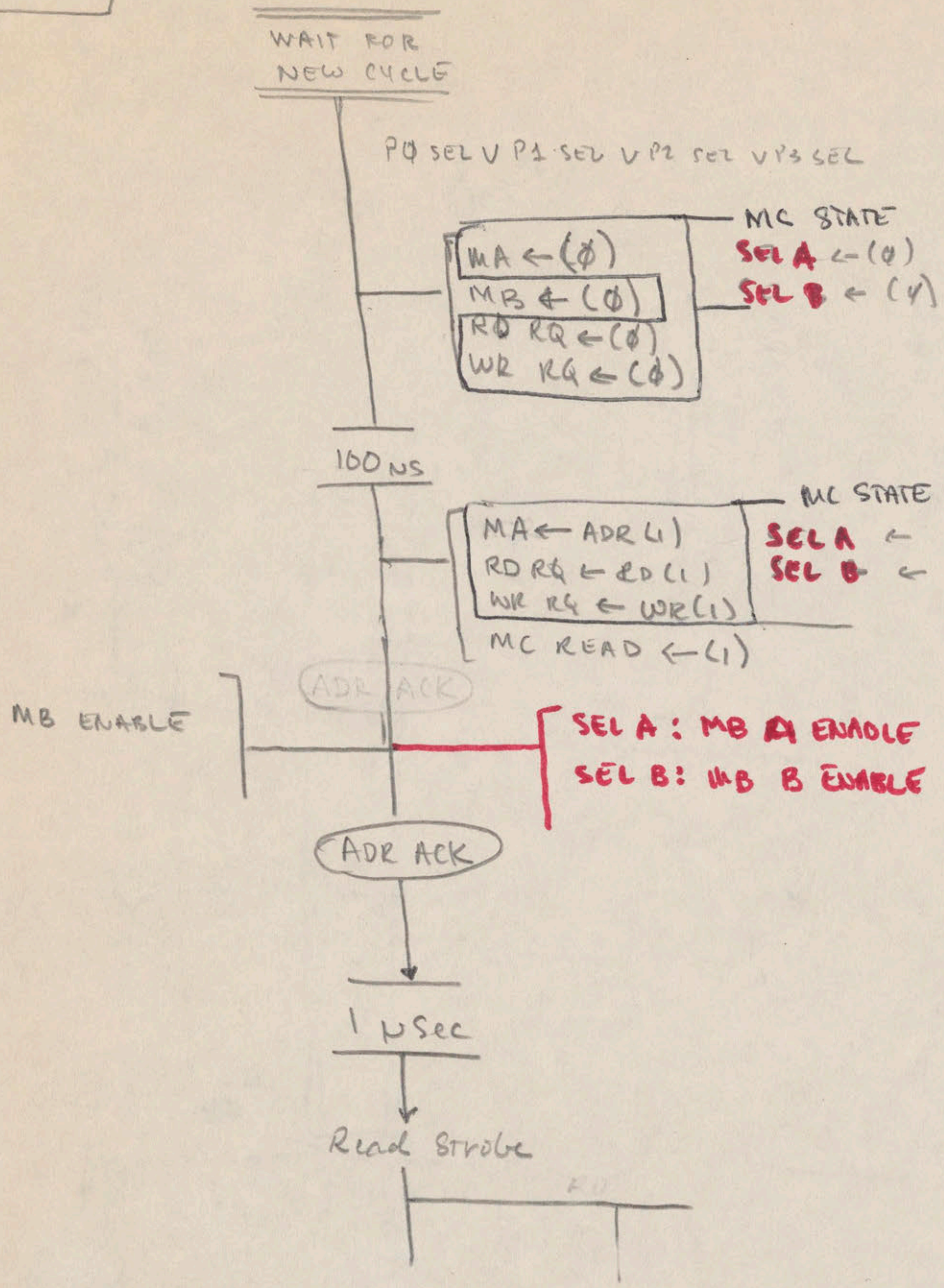
✓ MEM AC MEM

✓ MEM AC AC

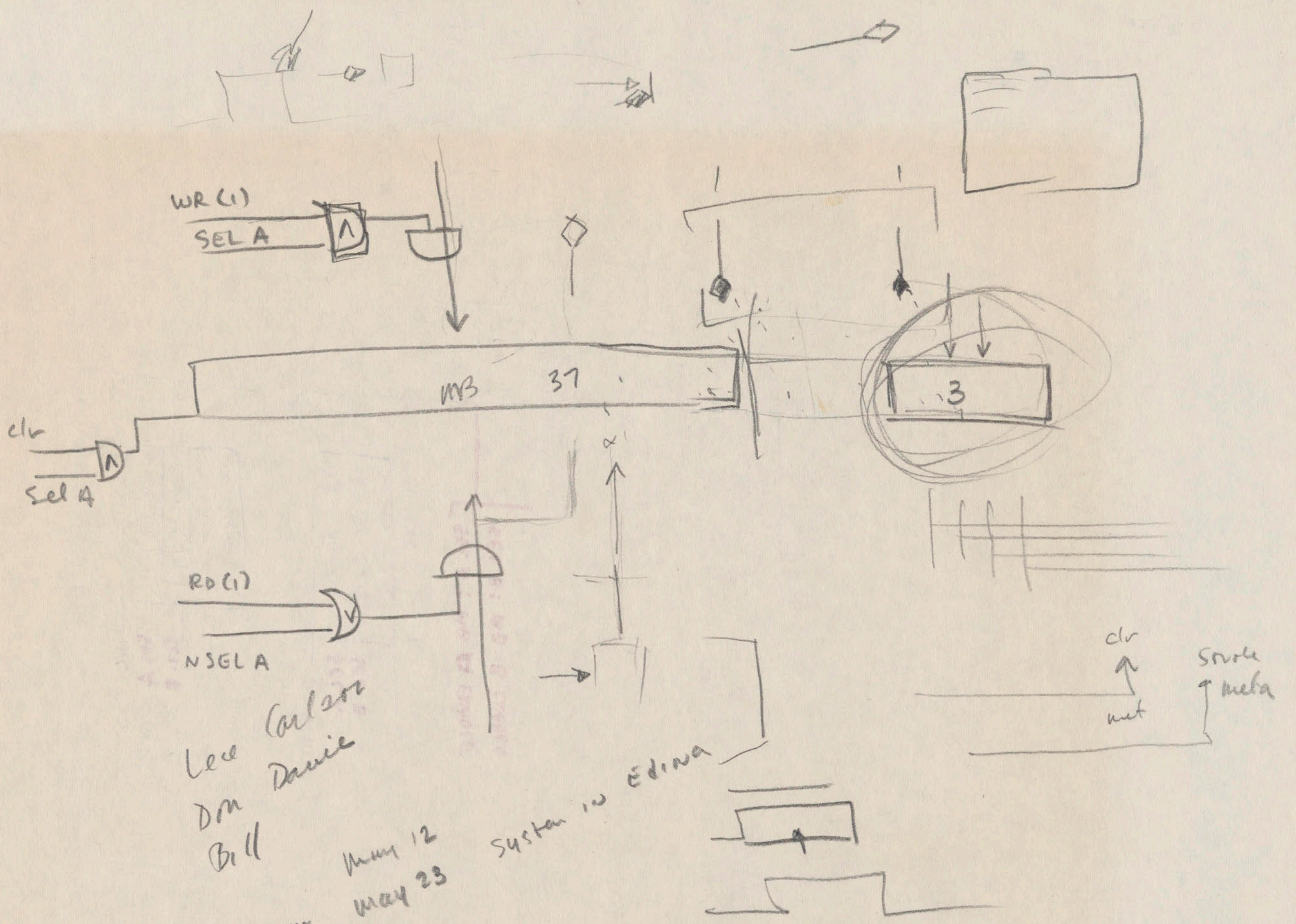
✓ ACCP

IR ACDM

A B



1. Stack
2. Delivery Date



Lee Carlson

Don Dawie

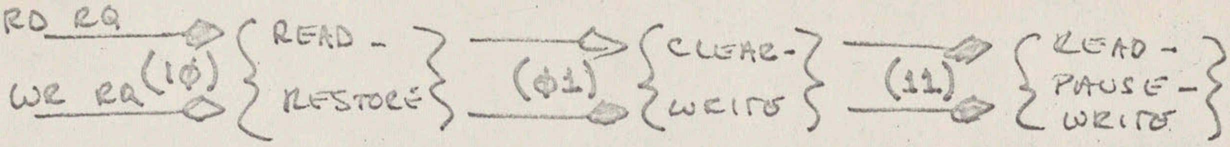
Bill

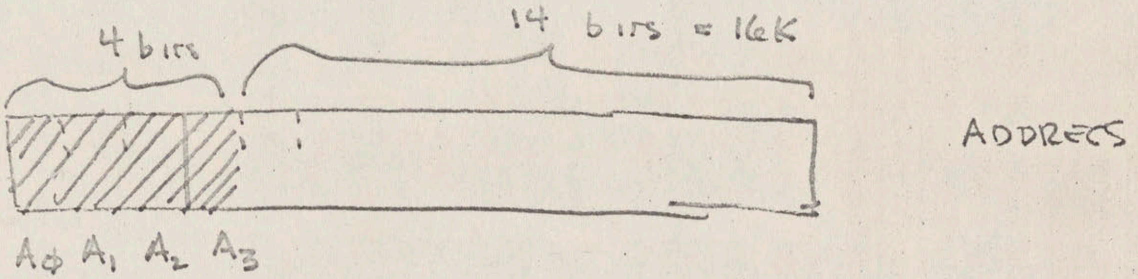
May 12

optim May 23

system in Edina

MEMORY CYCLES





CONDITION FOR CYCLE REQUEST

CYC RQ $= (A_\emptyset + A_1 + A_2 + A_3 + SM)$ and $\overline{\text{FM SEL}}$ and RQ

where SM implies Special MODE a "ONE"

$\overline{\text{FM SEL}}$ implies not MA 22-31 = ∅ or not Fast memory select

and RQ implies ~~that~~ ~~cycle~~ processor request is true

CONDITIONS FOR cycle being honored

Memory is not currently in a cycle

and CYC RQ is a "one"

and Processor has highest priority of those with CYC RQ

[A over B over C and D, C and D alternate]

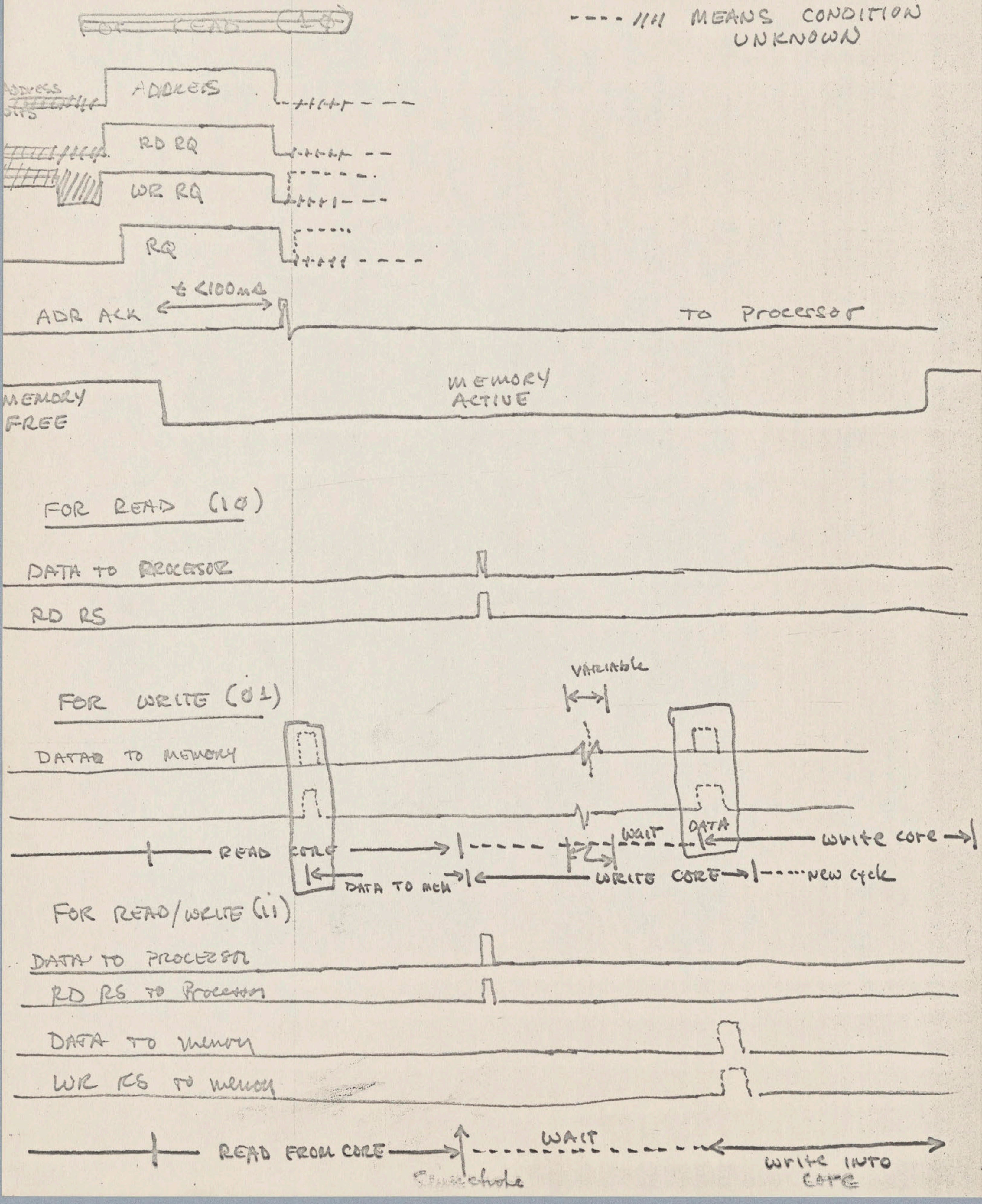
Steps to occur at The beginning of a memory cycle

1. Highest Priority Processor Determined and stored
2. Adress (18 bits), RD RQ, WR RQ strobed INTO Flip Flops
3. Send ADR ACK pulse to processor
4. begin cycle. if read (1∅) sense strobe → bus
   RD RS → Processor
   continue on to rewrite + end of cycle

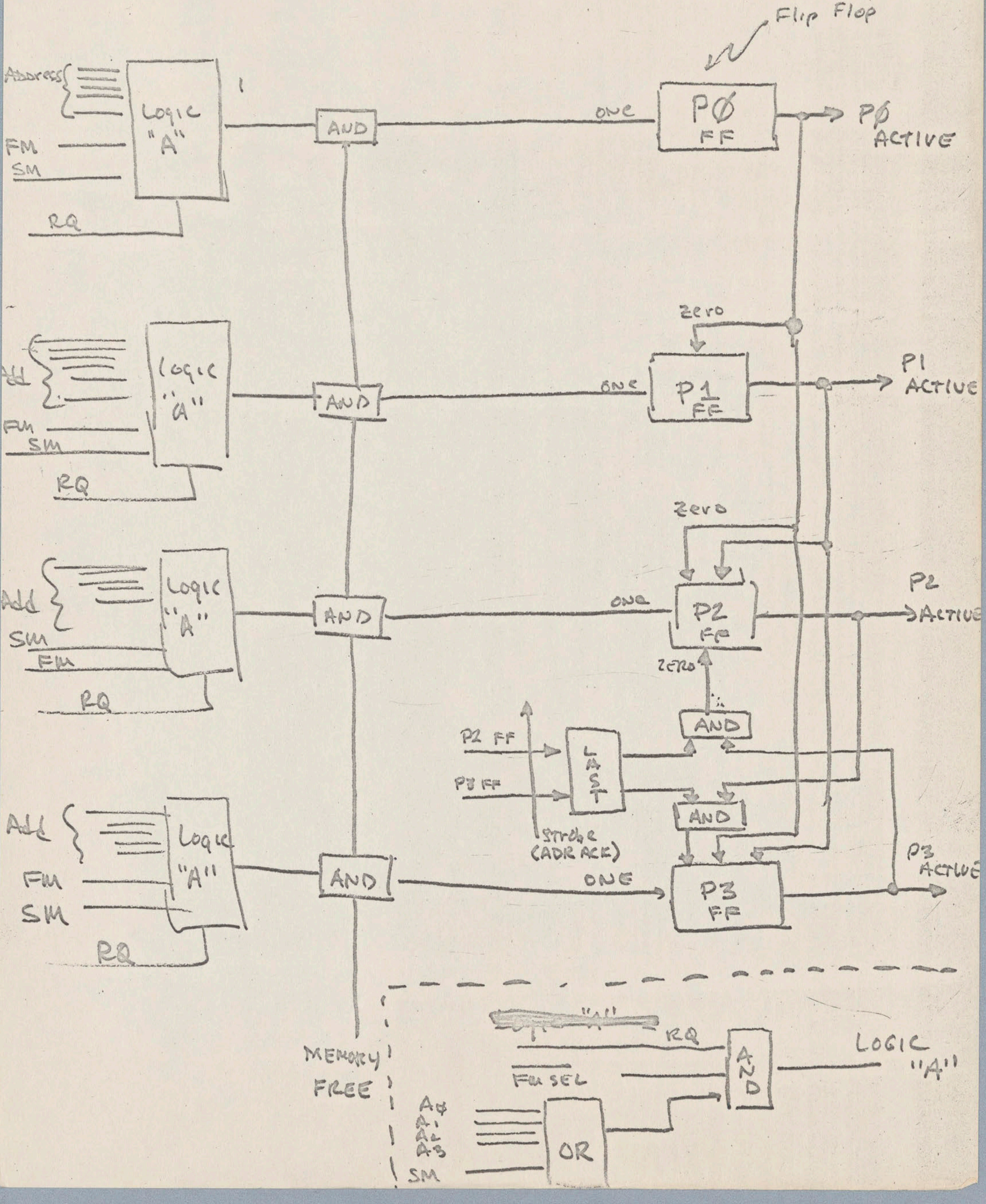
if write (∅1) no signals onto bus
input to MB open
If WR RS comes, then
write + finish cycle

if read/write (11) Sense strobe → bus
RD RS → Proc
WAIT with MB open
if WR RS
then write + finish cycle.

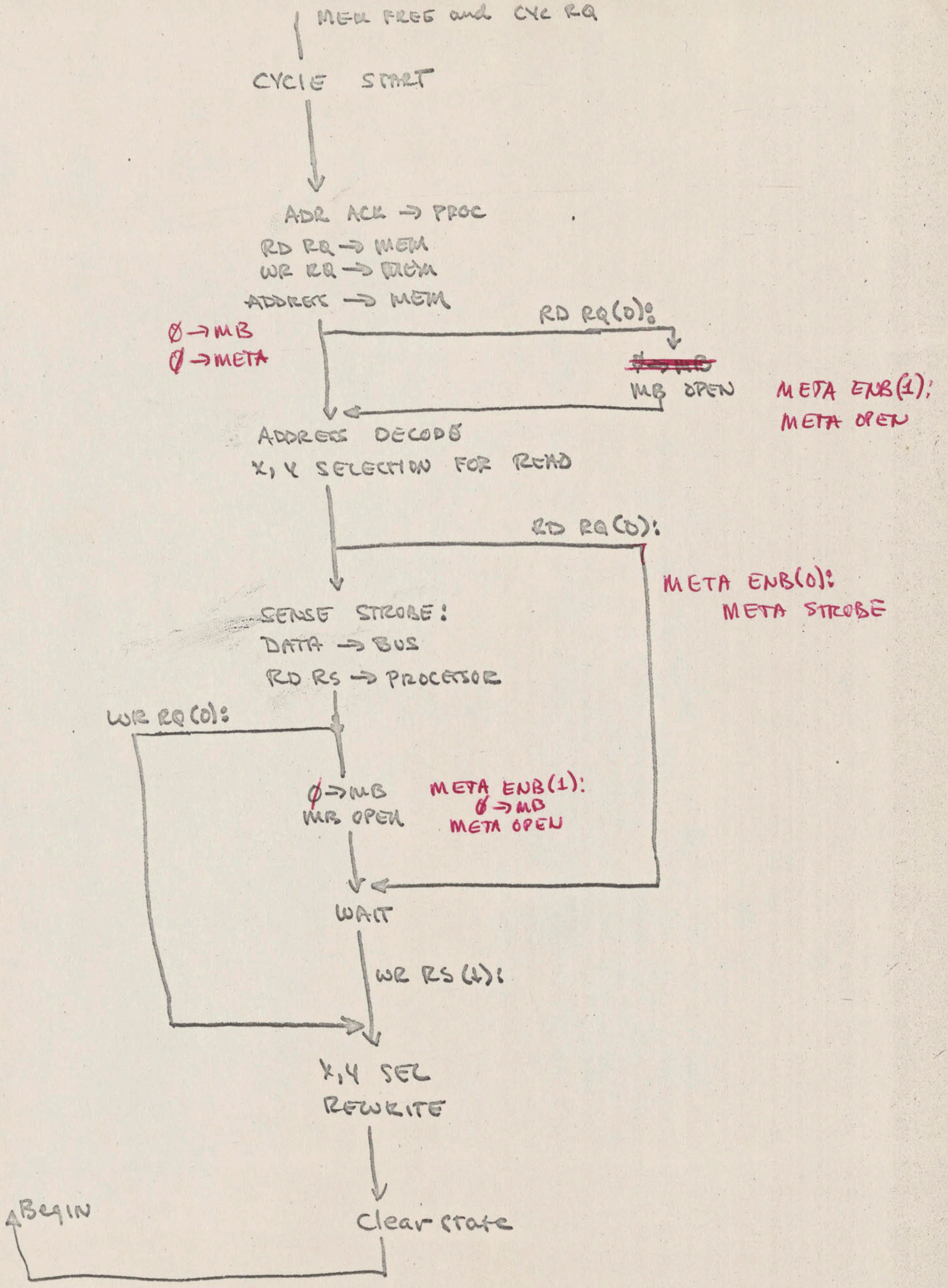
TIMING

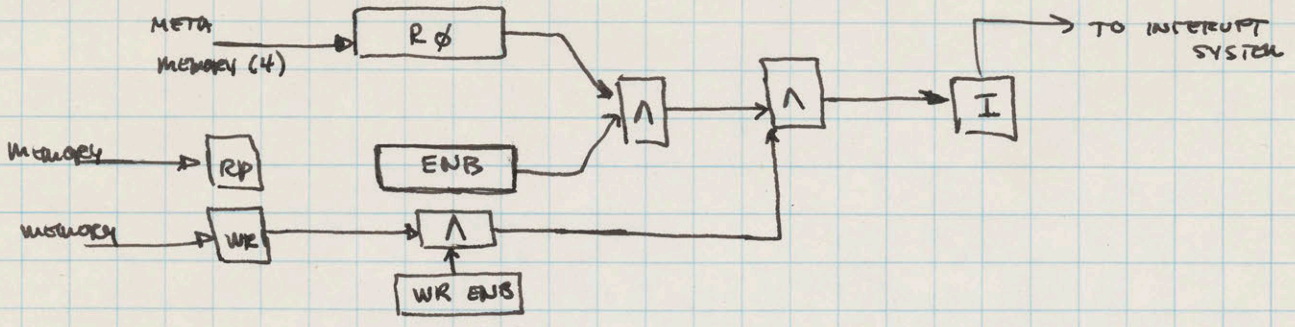

~~FOR READ (1Ø)~~
---- //// MEANS CONDITION UNKNOWN
ADDRESS BITS
ADDRESS
RD RQ
WR RQ
RQ
t <100ma
ADR ACK
TO Processor
MEMORY FREE
MEMORY ACTIVE
FOR READ (1Ø)
DATA TO PROCESSOR
RD RS
FOR WRITE (Ø1)
VARIABLE
DATAR TO MEMORY
WAIT
DATA
READ CORE
write core
DATA TO MEM
WRITE CORE
new cycle
FOR READ/WRITE (11)
DATA TO PROCESSOR
RD RS TO Processor
DATA TO memory
WR RS TO memory
READ FROM CORE
WAIT
Strobe
write INTO CORE

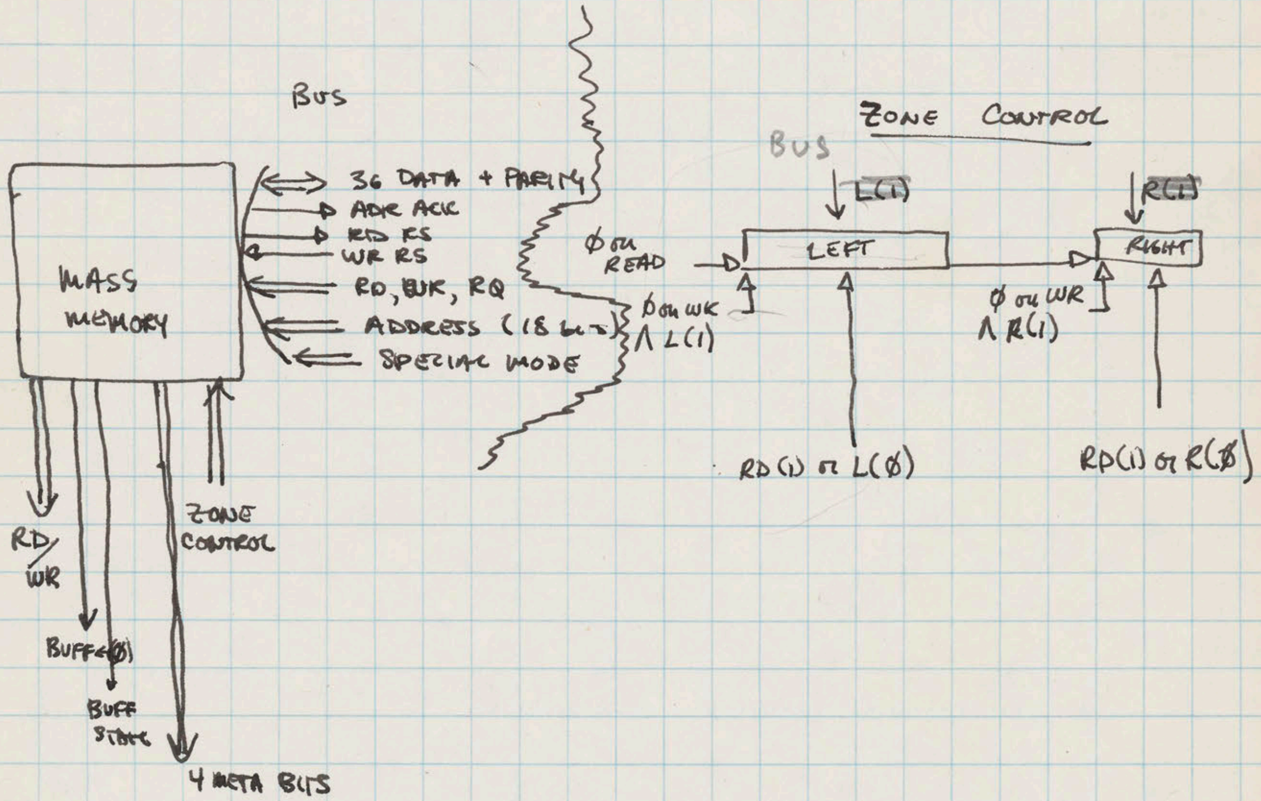
NOTE THAT DURING A WRITE ONLY (Ø1) cycle the DATA and WR RS may arrive from the processor any time after the ADR ACK, even before the memory has finished the clear-to-zero or read portion of a cycle. The memory Buffer should be open to the bus from the time the Address acknowledge (ADR ACK) occurs, and the memory should store the WR RS for continuing at the end of the read cycle.



Flip Flop
Address
Logic "A"
FM
SM
RQ
AND
ONE
PØ FF
PØ ACTIVE
zero
Add
Logic "A"
FM
SM
RQ
AND
ONE
P1 FF
P1 ACTIVE
zero
Add
Logic "A"
SM
FM
RQ
AND
ONE
P2 FF
P2 ACTIVE
ZERO
AND
P2 FF
P3 FF
LAST
Strobe (ADR ACK)
AND
Add
Logic "A"
FM
SM
RQ
AND
ONE
P3 FF
P3 ACTIVE
MEMORY FREE
~~"A"~~
RQ
FM SEL
AND
LOGIC "A"
Aφ
A1
A2
A3
SM
OR


MEM FREE and CYC RQ
CYCLE START
ADR ACK → PROC
RD RQ → MEM
WR RQ → MEM
ADDRESS → MEM
∅ → MB
∅ → META
RD RQ(0):
~~∅ → MB~~
MB OPEN
META ENB(1):
META OPEN
ADDRESS DECODE
X, Y SELECTION FOR READ
RD RQ(0):
META ENB(0):
META STROBE
SENSE STROBE:
DATA → BUS
RD RS → PROCESSOR
WR RQ(0):
∅ → MB
MB OPEN
META ENB(1):
∅ → MB
META OPEN
WAIT
WR RS(1):
X,Y SEL
REWRITE
Clear state
Begin





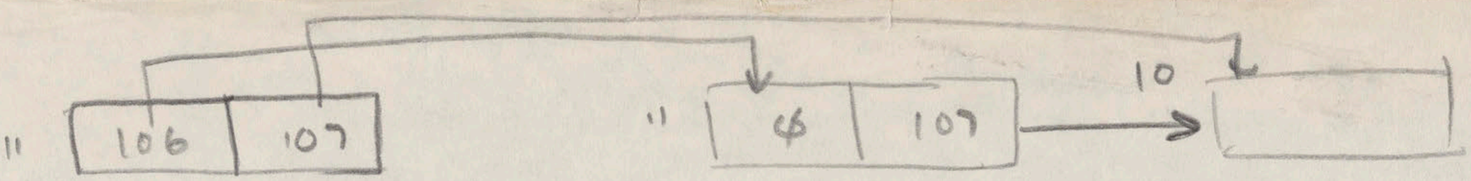
READ AND WRITE FLIP/FLOPS EXTERNALLY AVAILABLE

BUFF <- (∅) " "

← BUFFER GETS SET " "

4 META BITS

→ ZONE CONTROL (3 bits vs. 37 bits)



00 107 ∅

01 ∅ 106

106 107

00 107 105

01 105 ∅

01 0 107

∅ (∅)105

M↔AKRT
MLTL↔MRT
M↔AK

∅ (105)106

∅ (106)∅

1 (∅)106

MRTL↔MRT
M↔AK RT
M↔AK

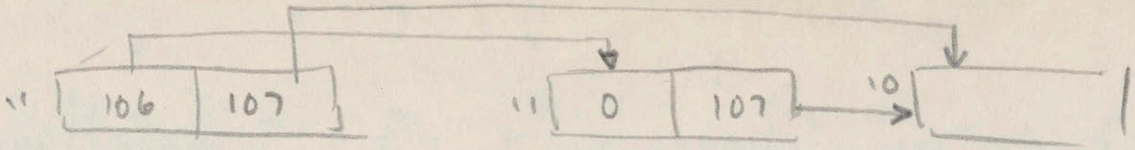
(107)106

0 (106)107

AR	105	106	∅	107	106	107	105
MQ	∅	105	106	106	105	105	0

MB↔AK RT

MQ ↔ MB RT

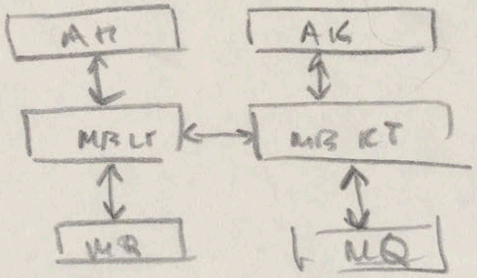


00 0 107

00 105 107 00

01 165 ∅

01 ∅ 107



(0) 105

(105) 106

(106) ∅

(106) 107

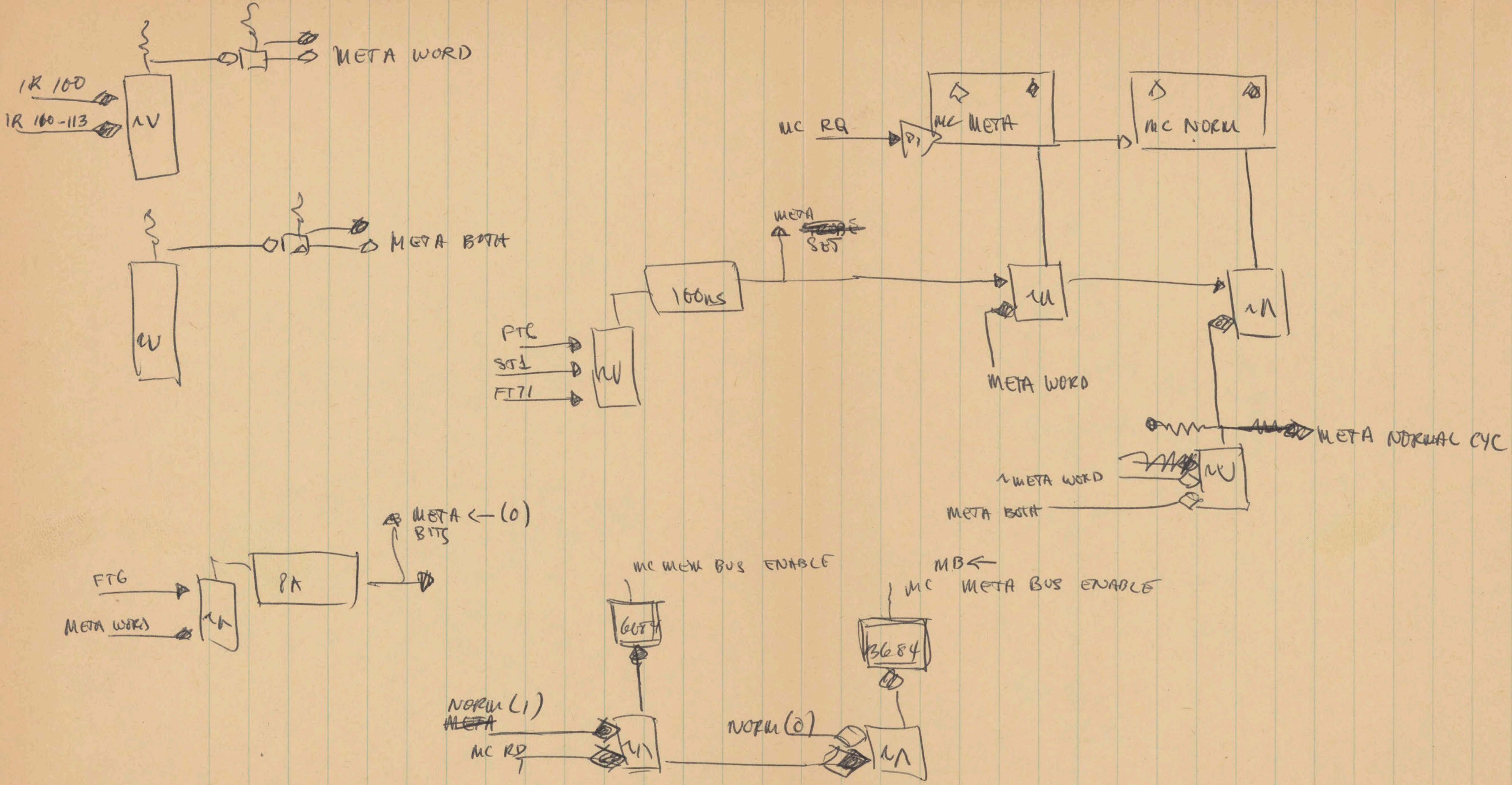
(106) 105

THE WORST OF THE WORST

```
1.       JCL      JUMP AND CLEAR AC
2.       JCDR     JUMP AND TAKE CDR
3.       SOBJN    SUBTRACT ONE FROM BOTH AND JUMP IF NEGATIVE
         SOBJP
4.       SOBS..   SUBTRACT ONE FROM BOTH AND SKIP (LE,E,A,GE,G,L,N)
         AOBS..
5.       AOBJE    ADD ONE TO BOTH AND JUMPE IF EQUAL(TO ZERO)
         AOBJN,SOBJE,SOBJN
6.       SKIPI    EFFECT ADD GOES TO AC
7.       SKIPM    AC GOES TO MEMORY
10.      XCT N,   XCT BUT PC+N => PC
11.      PUSHI    PUSH IMMEDIATE
12.      LCAI..   COMPARE WITH LEFT HALF
13.      RCAI..   COMPARE WITH RIGHT HALF
14.      DBP      DECREMENT BYTE POINTER
         LDBD,DPBD
15.      DMOVE    DOUBLE WORD MOVE OPS
16.      RBLT     REVERSE BLT,(GOES FROM HIGH TO LOW)
17.      RBLKI    SAME
         RBLKO
20.      MARK     HARDWARE MARK INSTRUCTION FOR LISP
21.      NORM     36 BIT NORMALIZE
22.      CNT      COUNT BITS INSTRUCTION
23.      POPJ P,N         USE ADDRESS TO POP AC'S,SKIP OR OTHER RANDOMNESS
24.      LJMP     JUMP THROUGH LEFT HALF
25.      ??       JUMP IF EQUAL,OTHERWISE TAKE CAR INTO AC+1
26.      XMT      MEMORY TO MEMORY TRANSFER
27.      MTP N,ADR(P)     TRANSFER ADR TO -N(P)     (GROAN)
30.      TM()()   TEST BUT RESULTS TO MEMORY
31.      LAOS     AOS LEFT HALF
         LSOS
         RAOS
         RSOS
32.      MPUSH    C(AC)=>C(C(E)+1=>C(E))
33.      MPOP     C((C(E)-1=>C(E))+1)=>C(AC)
34.      MPUSHJ   C(PC)=>C(C(E)+1=>C(E))  E+1=>C(PC)
35.      MPOPJ    C((C(E)-1=>C(E))+1)=>C(PC)
36.      HRB      HALF RIGHT TO BOTH HALVES
         HLB
37.      CLEARM AC,       OLD C(E) GOES TO C(AC) IF AC=/=0

         MODE WHERE INDIRECT BIT MEANS BYTE OPERAND

38.      JL6      JUMP IF AC+1= 0
                  Otherwise clear AC AND LSHC AC,6 until AC ≠ 0
39.      FILL     CORE
4Ø.      FEDCBA   GET BUSTED BY COP, WHO COMPLIMENTS
                   BOTH OF YOU. YOU ALWAYS SKIP OUT.
41.      GVM      GROVEL & MUNG
                  Replaces each word in memory by the word it addresses,
42. SUR(-,I,M,B) } subtract reversed (ie, with AC negated)
    FSR(-,  M,B) }
43. AOC, SOC  add, subtract ones-complement.
```



IR 100
IR 100-113
META WORD
META BOTH
MC RQ
P1
MC META
MC NORM
META
~~STROBE~~
SET
100ns
FTC
ST1
FT71
META WORD
META NORMAL CYC
~META WORD
META BOTH
META ← (0)
BITS
FTG
META WORD
8A
MC MEM BUS ENABLE
MB ←
MC META BUS ENABLE
6684
3684
NORM (1)
~~META~~
MC RD
NORM (0)

AC IMM = IR METABM

INITIAL REGISTERS   AR = QUOTE AC
                         MB = C(E)

INITIAL GATES   FC(E) PSE
                     META WORD
                     E LONG
                     AC IMM

ETOA

ETOA   AR(J) ⟷ MB(J)   *

ET1   MB ← AR(0)
           ACBM COM AR ← MB(+)
           ACBM SET ! AR ← MB(1)

ET3 —

ET4   MB(J) ⟷ AR(J)

ET 5   AR COM

ET 6   ACBM CL ! MB ← AR(0)

ET7   AR COM

ET8

ET9   MB(J) ⟷ AR(J)
           ACCP ET AC TEST ! PC+1

ET

ET 10

FINAL gate   , SAC INIT

MODS TO IR

IR ACBM at!

MB ← AR(0) (ET1)

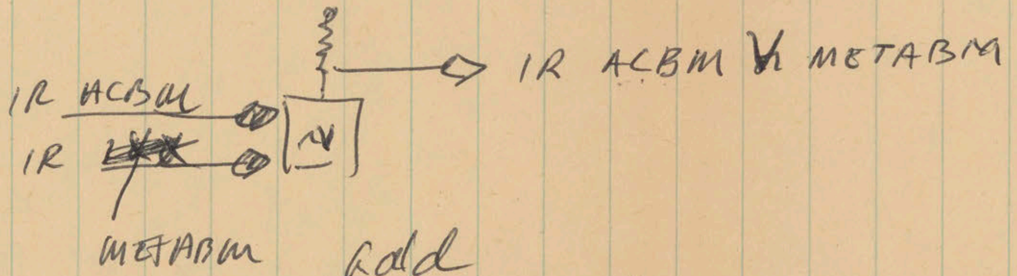
MB(J) ↔ AR(J) (ET4)

AR com (ET5)

AR com (ET7)

MB(J) ↔ AR(J) (ET9)

BECOMES



add

AR(J) ↔ MB(G) (ET0)

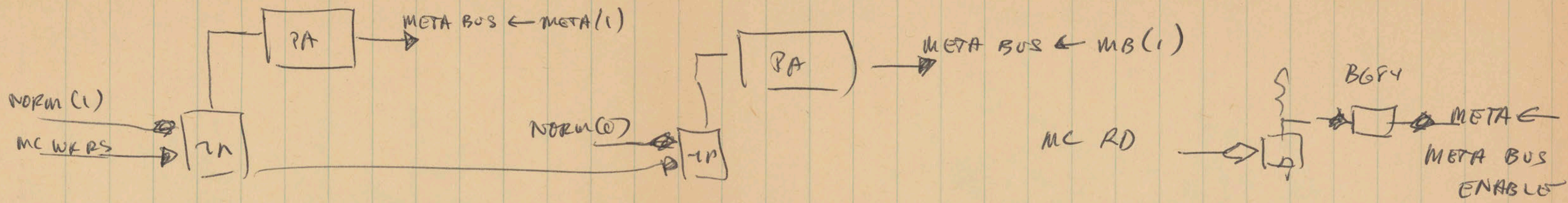
FC(E) PSE

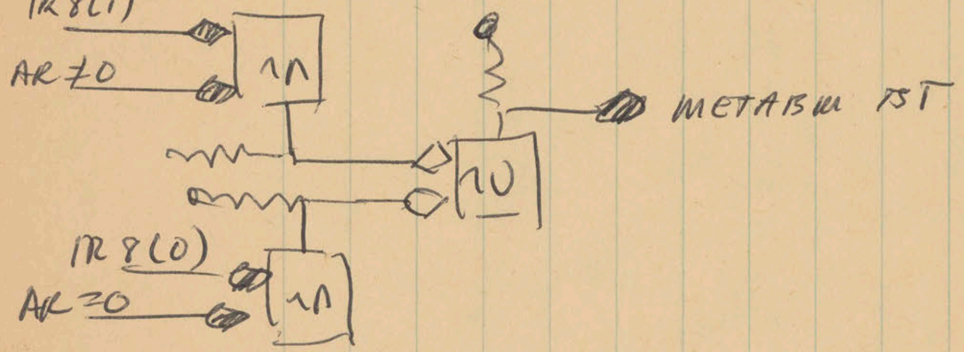
META WORD

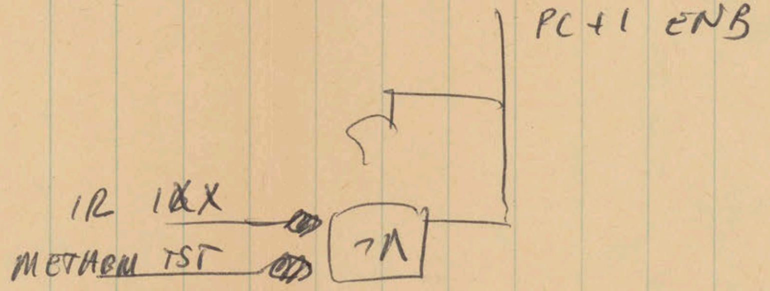
E LONG

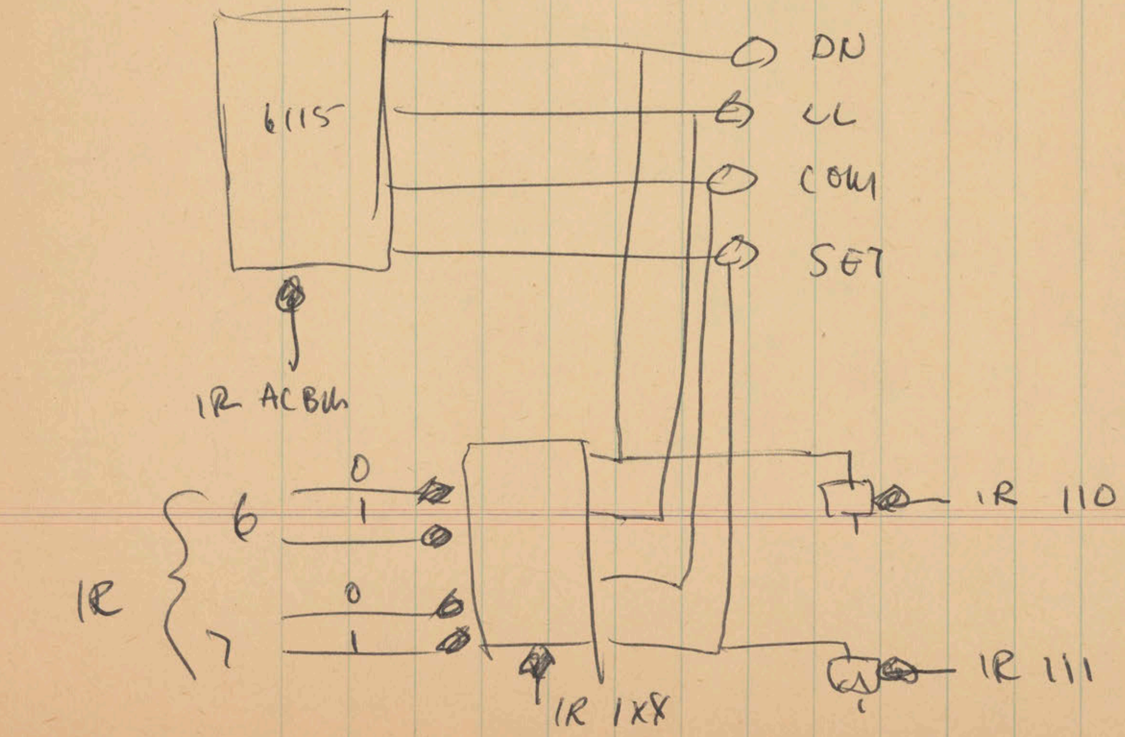
SAC INH

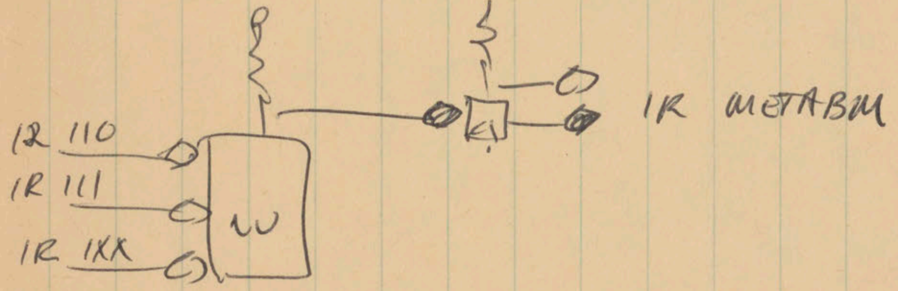
from → IR ~~ACBM~~ METABM


PA
META BUS ← META(1)
PA
META BUS ← MB(1)
NORM (1)
MC WKRS
NORM(0)
MC RD
BGF4
META ←
META BUS ENABLE


IR 8(1)
AR ≠ 0
¬∧
METABM TST
¬∨
IR 8(0)
AR = 0
¬∧


PC+1 ENB
IR 1XX
METABM TST
¬∧


6115
DN
LL
COM
SET
IR ACBM
0
1
6
IR
0
1
7
IR 1XX
IR 110
IR 111


IR 110
IR 111
IR 1XX
¬∨
IR METABM

	IR 112 LOAD	IR 113 STORE
INITIAL GATES	META WORD FC (E) FAC INH	META WORD
FINAL GATES	—	SC (E)

1935-W-Ø-M-P

ADDED INSTRUCTIONS. AND MODS

1. JCL JUMP AND CLEAR ACCUMULATOR
2. SKIPI SKIP IMMEDIATE
3. SKIPM SKIP TO MEMORY
4. (MOD) FOR MOVE, MOVEI, MOVEM, MOVES
GIVES ET1 INHIBIT ET0 → ET1Ø:
5. MODIFY PI SYNC SO THAT 200 us not wasted.
6. PUSHI Push Immediate
7. SOBJN
SOBJP Subtract one both & jump {complement AOBJN
10. RBLT Reverse BIT
11. CONS INSTRUCTION (DONE RIGHT) (ANY AC WITH ANY AC)
12. GC INSTRUCTION
13. JCR JUMP AND TAKE CDR OF AC
14. HCAI{XX} LEFT 1/2 OF AC FLUSHED BEFORE COMPARE
15 DMOVE
DMOVEM DOUBLE LOAD + DOUBLE STORE
16 META BIT TRAP ON INSTRUCTION FETCH
WRITE CYCLE
ANY REFERENCE
17 MA=-1 GIVES NON-EX-MEM
18 SWITCH THAT SENDS OLD PC TO MI WHENEVER ~~FA~~ TRANSFER OCCURS

(OVER)

19. NORMALIZE INSTRUCTION

20 COUNT BITS INSTRUCTION

21 POP THAN POPJ INST.

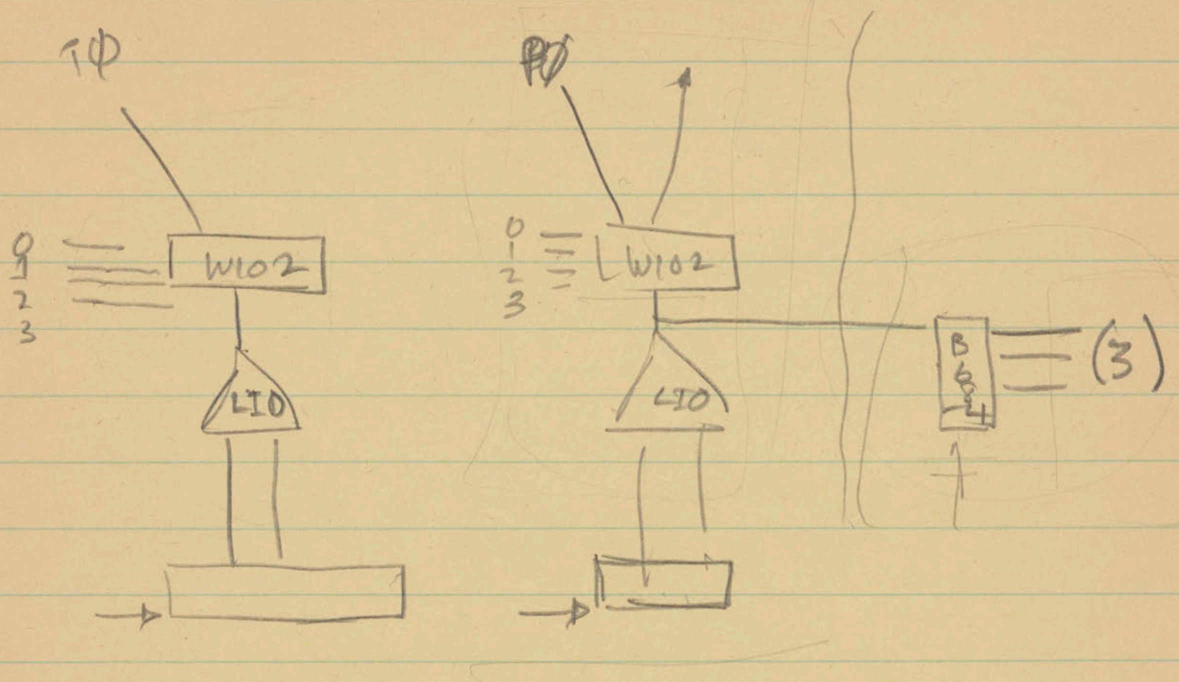
22 JUMP THRU LEFT HALF

23. DECF BYTE

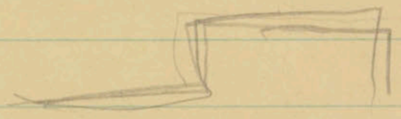
24. OP WITH BYTE ADR. AOS, XOR, CAI

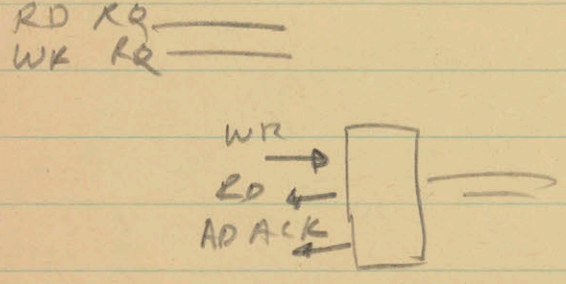
25. JUMPL if zero otherwise take cor AC+1

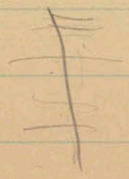
26.



1. T Clock







(2) ZONE CONTROL BITS 37 bits VS 3 bits
"0" Zero (GND) ENABLE WRITE IN THIS HALF
"1" one (-3v) READ/RESTORE on section

IF ZONE CONTROL IS A "ONE"

1. CORRESPONDING HALF OF WORD DOES NOT GET CLEARED DURING THE BEGINNING OF SECOND HALF OF READ/PAUSE/WRITE.
2. DATA FROM STACK AT THE NORMAL READ TIME IS GATED INTO CORRESPONDING HALF OF MEMORY BUFFER.
3. IT IS ALLOWABLE TO HAVE DATA REGISTER INPUTS FROM BUS ENABLED EVEN IF ZONE BIT IS A "ONE"
4. Data will not be transfered onto bus if A WRITE ONLY cycle occurs, even if corresponding zone bit is a one.

EXTRA MONITOR, bits

APPEAR AS LEVELS (-3 "ONE")
3 DATA BITS
(2) RD RQ - WR RQ
(4) Processor select.

Come out on Single cable through BC84 cards

as .

1 → Pulse appear in at normal READ ~~RESTART~~ time.
Restart

May 28, 1964
A. Kotok

PDP-6 MEMORY BUSS SYSTEM

One Memory can be connected to as many as four processors via the memory buss system. Figure 1, attached, shows a typical system expanded to full capacity with four processors (hence, four busses) and with four Memories Type 163C.

Each buss consists of four flat 18-conductor ribbon coax cables. These cables carry all the necessary logic signals between a processor and whatever number of Memories (up to sixteen) there are in the system. The signals carried by these cables are described in "Description of Logic Signals", attached.

The buss cables are actually segmented. That is, the four cables do not run continuously from the processor to each of the memory controls. Four cables run from a processor to 4 connectors in one memory; four additional connectors carry the 4 x 18 lines out of this memory to four additional cables and on to the next memory. The buss continues in like manner to the last memory. Figure 2 illustrates the buss system.

The buss system connectors plug into mating connectors on the handle end of the special interface modules in the Memory. The 4 cables constituting the buss each plug into one of 4 corresponding interface modules. The continuation of the buss to the next memory control is from another connector on the handle end of each of the same interface modules, as shown in Figure 2. The special interface modules are:

cable I	Type 1664
cable II	Type 1665
cable III	Type 1665
cable IV	Type 1665

Thus, the number of interface modules in each memory is equal to four times the number of processors.

Cables I and II terminate with a Type 1032 connector at the processor. This connector plugs into a Type 1901 mounting panel in the processor in the same manner as a standard DEC system module, except that it mates with a male connector, Hence, the female connectors that are normally provided with a Type 1901 mounting panel must be removed from the 2 positions into which cables I and II will plug. They are replaced with two 22-pin male Methode connectors. Methode part number 130-12-MD6225. These connectors must be installed on 1/8" standoffs. Cables III and IV terminate with Type 1031 right angle connectors at the processor. These connectors mate with the connectors on the handle end of Type 1665 modules which must be used in the processor to handle MB data.

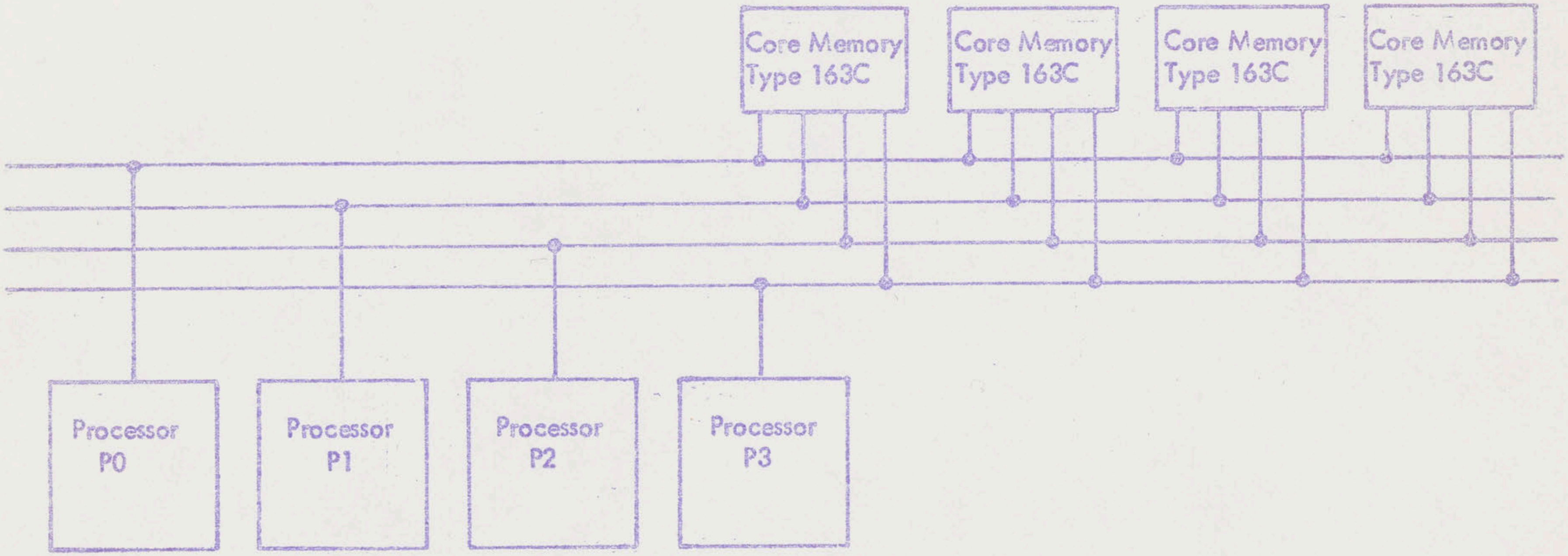

Core Memory Type 163C
Core Memory Type 163C
Core Memory Type 163C
Core Memory Type 163C
Processor P0
Processor P1
Processor P2
Processor P3

Figure 1

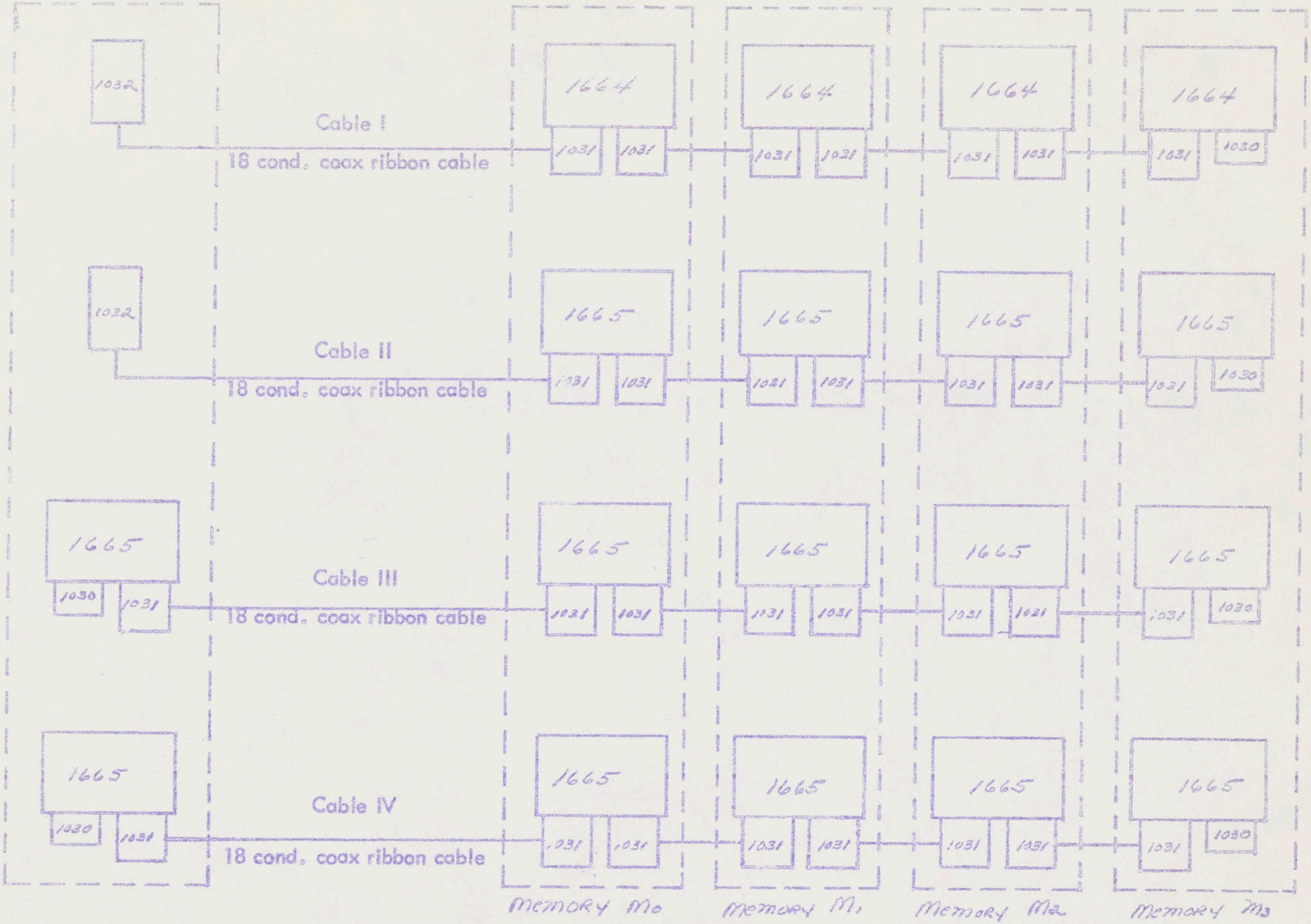


Figure 2

Memory Buss System For One Processor

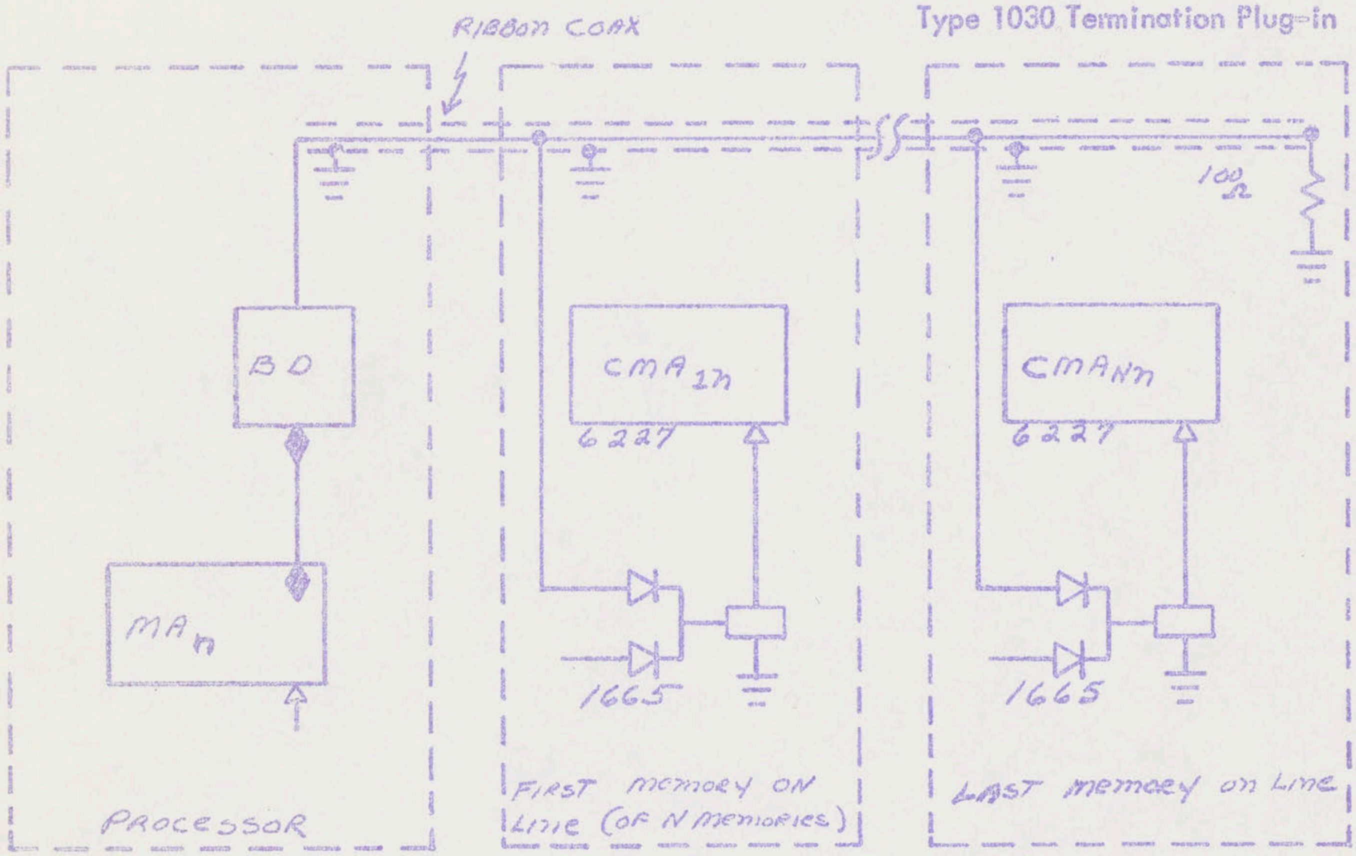
A special mounting panel, the Type 1933, can be used in the processor to permit plugging of the Type 1665 module. This mounting panel is twice the height of a standard mounting panel and has 2 positions for inserting modules of the Type 1665 size. The Type 1933 panel has, in addition, positions for plugging in 44 standard DEC system modules. Two of these 44 positions could be used for plugging cables I and II.

Teminating plug-ins, type 1030, are used on the handle ends of the interface modules of the last memory control on the buss. These plug-ins each contain 18 100-ohm resistors to provide proper matching to the coax lines.

Figure 3, "14 Memory Address Lines" and Figure 4, "36 Memory Buffer Lines" illustrate representative circuitry used with the buss system. The circuit requirements for the buss system are further specified in "Bus System Circuit Requirements", attached. Three tables are also included which list the coax wire assignments for cables I, II, III and IV.

Processor Core Memory Control Interface

14 Memory Address Lines



Terminating Resistor located on Type 1030 Termination Plug-in

14 MA lines are in coax cable II which plugs into Processor with a Type 1032 connector.

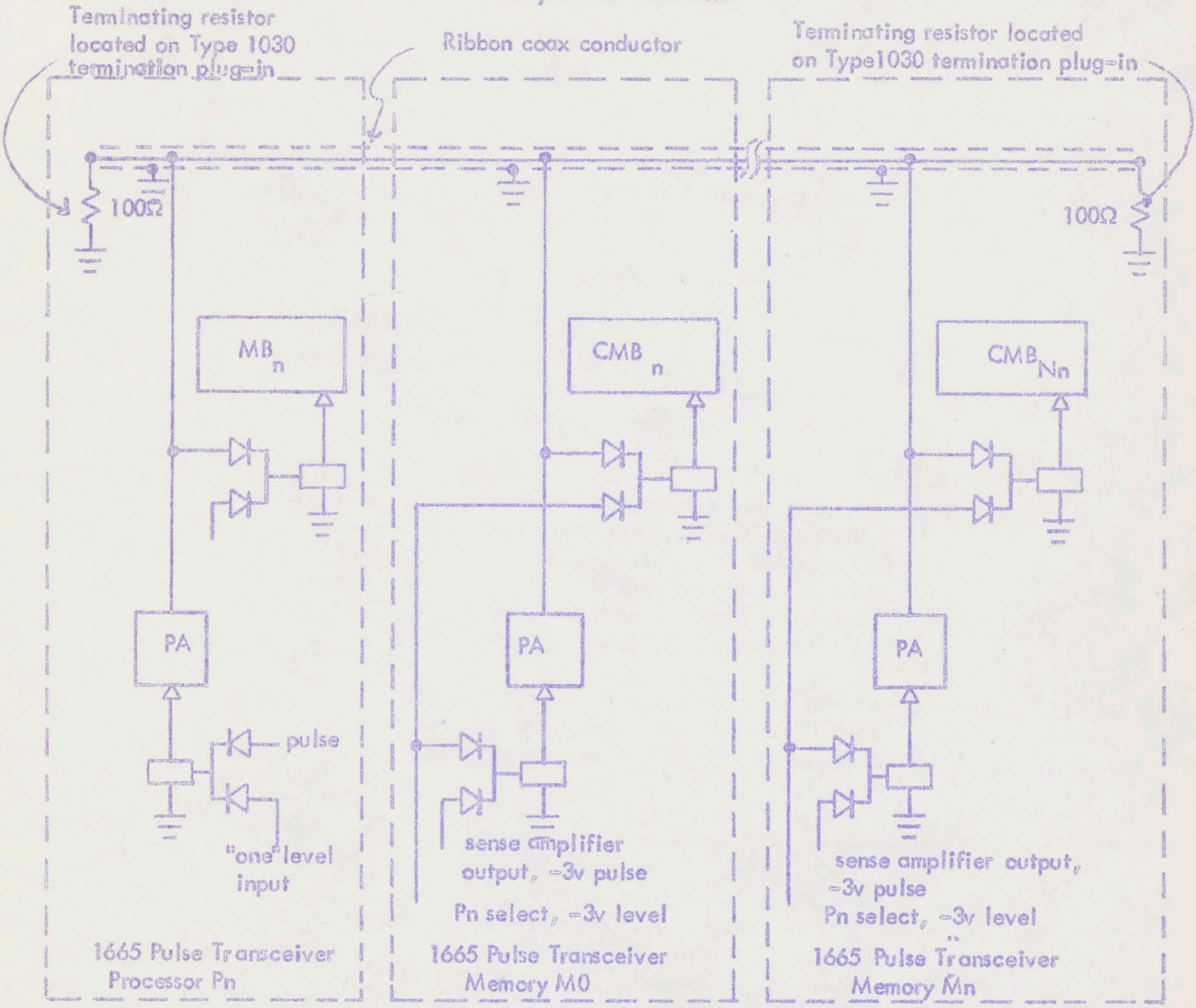
Connector Pin Assignments

MA. Bit	22	23	24	25	26	27	28	29	30	31	32	33	34	35
Pin No.	B	C	D	E	F	H	K	L	M	N	P	R	T	U

Figure 3

Processor Core Memory Control Interface

36 Memory Buffer Data Lines



Terminating resistor located on Type 1030 termination plug-in
Ribbon coax conductor
Terminating resistor located on Type1030 termination plug-in
100Ω
100Ω
MB n
CMB n
CMB Nn
PA
PA
PA
pulse
"one" level input
sense amplifier output, -3v pulse
Pn select, -3v level
sense amplifier output, -3v pulse
Pn select, -3v level
1665 Pulse Transceiver Processor Pn
1665 Pulse Transceiver Memory M0
1665 Pulse Transceiver Memory Mn

Cable III/IV 36 MB lines are in coax cables III and IV which plug into processor with Type 1031 connectors.

Connector Pin Assignments

MB Bit	0/18	1/19	2/20	3/21	4/22	5/23	6/24	7/25	8/26	9/27	10/28	11/29	12/30	13/31	14/32	15/33	16/34	17/35
Pin No.	B	C	D	E	F	H	K	L	M	N	P	R	T	U	V	W	X	Y

Figure 4

Core Memory Type 163C Interface With Processor

Description of Logic Signals

Signal — Description

RQ
-3V level
Cable 1
Pin F

A processor requests a memory cycle by asserting its RQ line. If a memory cycle is not already in progress and if no other processor of higher priority is simultaneously requesting a memory cycle, a memory cycle will immediately begin. If another memory cycle is in progress, that cycle must complete before the requested cycle can begin. If another processor of higher priority is requesting a memory cycle, that processor's cycle will be executed first. The RQ signal must remain present at least until an ADRS ACK pulse is returned to the processor generating the RQ, at which time the requested cycle has begun. The RQ signal must be removed immediately after the ADRS ACK pulse is received at the processor, or a second cycle may begin.

ADRS ACK
-3v pulse
Cable 1
Pin B

The return of an ADRS ACK (address acknowledge) pulse is to a processor indicates that the memory cycle that is being requested by that processor has been started and that the specified address is being accessed. If the request is honored immediately, that is, if no other cycle was being executed when the request was made, the ADRS ACK pulse will be received by the processor approximately 200 nanoseconds after the RQ signal is generated.

RD RS
-3v pulse
Cable 1
Pin C

An RD RS (read restart) pulse is returned to the processor at the time that the sense amplifiers are strobed. The RD RS pulse is received by the processor approximately .5 microseconds after the AD RS ACK pulse.

WR RS
-3V pulse
Cable 1
Pin D

A WR RS (write restart) pulse is required by the Memory Control whenever a split cycle is being executed; that is whenever both RD RQ (read request) and WR RQ (write request) are asserted. In split cycle operations, the write cycle will not start until WR RS occurs. WR RS can be sent from the processor as soon as ADRS ACK is received by the processor, or anytime thereafter. However, if WR RS is sent earlier than the completion of the read portion of the memory cycle, that is, earlier than approximately 1 microsecond after the ADRS ACK pulse is received, the write portion of the cycle will begin immediately upon completion of the read portion of the cycle. ~~If RD~~ RQ and WR RQ are both asserted, the memory timing chain will stop upon completion of the read portion of the cycle providing WR RS has not yet been received. Then the write portion of the cycle will begin immediately upon occurrence of the WR RS signal. The WR RS pulse should not be sent to the Memory Control at all during read only (RD RQ only asserted) or write only (WR RQ only asserted) cycles.

Corrected

Signal | Description

RD RQ
-3v level
Cable II
Pin V

A processor requests a word from memory by asserting the RD RQ line. If a memory cycle has been requested (RQ asserted) and RD RQ is not asserted, the word which is retrieved from memory during the read portion of the memory cycle will not be transferred to the processor via the 36 MB lines. To accomplish the transfer of the word to the processor, RD RQ must be asserted when RQ is asserted and can be turned off any time after ADRS ACK is received by the pr ocessor.

WR RQ
-3vlevel
Cable II
Pin W

A processor requests to write a word into memory by asserting the WR RQ line. If a memory cycle has been requested (RQ asserted) and WR RQ is not asserted, the word which is retrieved from memory during the read portion of the cycle will be read back into memory during the write portion of the cycle. If both RD RQ and WR RQ are asserted, the Core Memory Buffer in the Memory Control is cleared 0.20 microseconds after RD RS is received by the processors. The Core Memory Buffer will be ready to receive the new word from the 36 MB lines to be written into memory during the write portion of the cycle after another 100 µs. WR RQ must be asserted when RQ is asserted and can be turned off any time after ADRS ACK is received by the processor.

SEL
gnd
levels
Cable I
Pins K-Y

The six SEL (select) line/pairs specify which of up to sixteen memory modules are being addressed. These lines commonly carry the complementing logic levels of the five most significant bits of the memory address registerand whether this address is $<20_8$. The particular 6-bit code which a memory module responds to is determined by jumpers in the memory interface module Type 1664. The levels on the SEL lines must be established 50 nanoseconds before RQ is asserted and can be reset or otherwise changed any time after ADRS ACK is received by the processor.

ADRS
-3v level

The fourteen ADRS (address) lines specify one of 16, 384 memory locations to which access is to be made during the current cycle. Thus, in PDP-6, these lines transmit logic signals MA22(1) - MA25(1). The levels of the ADRS lines must be established by the time RQ is asserted and can be reset or otherwise changed any time after ADRS ACK is received by the processor.

MB
-3v pulse

Information is transferred between the processor and the Memory Control via the 36 MB lines. A word is retrieved from memory and is transferred from the memory to the processor over the 36 MB lines coincident with the occurrence of the RD RS pulse. If the processor is requesting a write only cycle (RD RQ not asserted) it must provide gating so that it does not respond to the signals on these lines. During

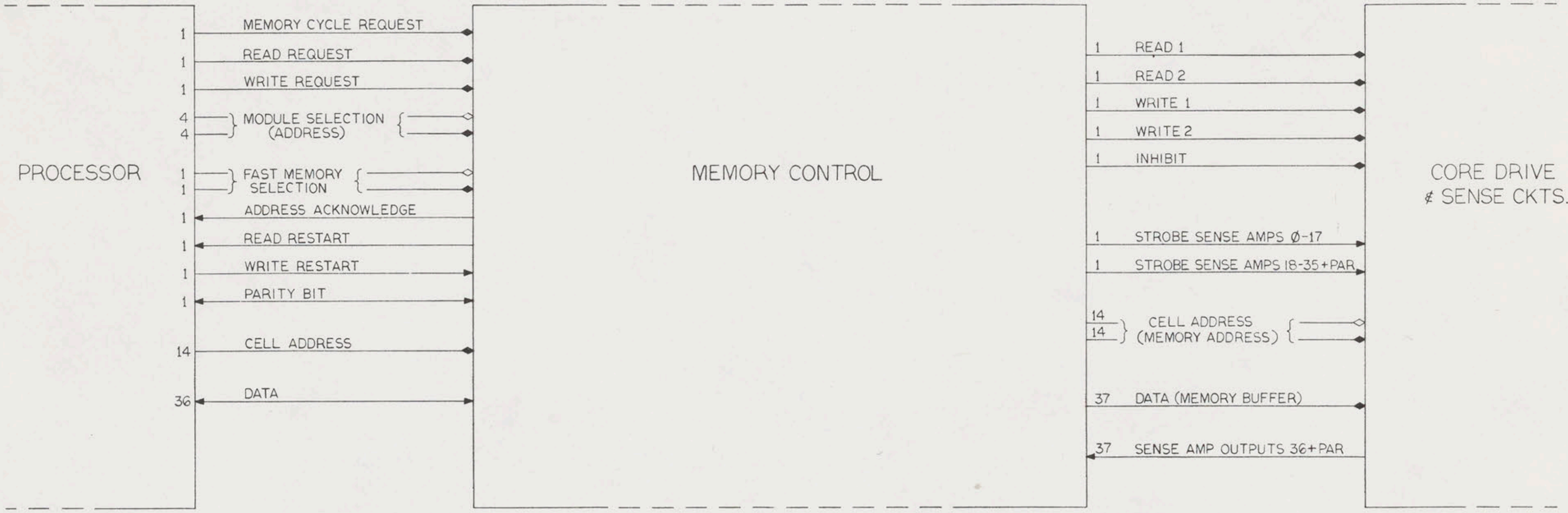
Signal	Description
	a read-write or a write only cycle, a word must be transferred from the processor to the m emory over the same lines not sooner than .1 microseconds after RD RS is received at the processor.

Buss System Circuit Requirements

1. All level inputs to the Memory should be driven with Type 6684 bus drivers, as shown in the accompanying diagram illustrating the interface circuits for the 14 memory address lines.

2. All pulse input to the Memory should be driven with a DEC 5 megacycle pulse amplifier, such as the Type 1607. All pulse lines are terminated at the Memory by 100 ohm resistors, as are all level lines.

3. All pulse outputs from the Core Memory Control must be terminated with a 100 ohm resistor to ground at the processor.



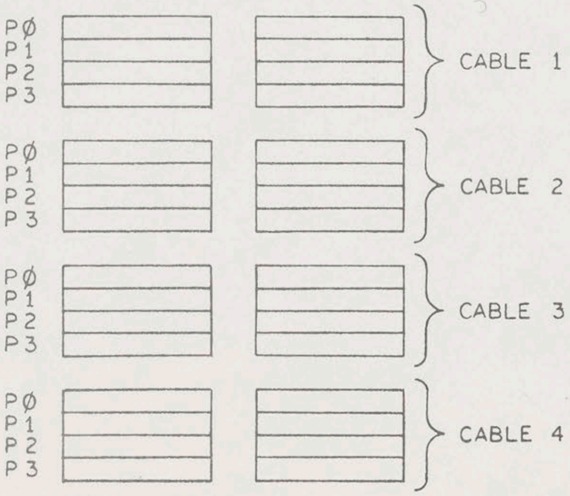
PROCESSOR/MEMORY INTERFACE

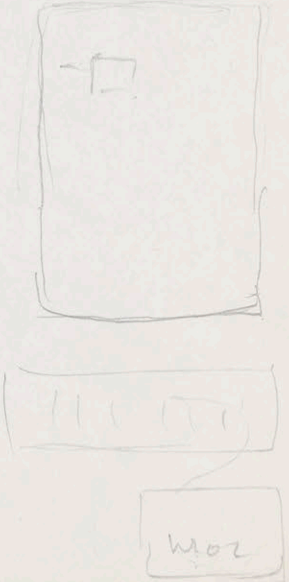
MEMORY CONTROL/DRIVE CKTS.
INTERFACE

MEM. BUS.

PIN	MEM CABLE #1	MEM CABLE #2	MEM CABLE #3	MEM CABLE #4
A	GND	GND	GND	GND
B	ADDR ACK →	MA 22 (1) —◆	MB Ø (1) →	MB 18 (1) →
C	RD RS →	MA 23 (1) —◆	MB 1 (1) →	MB 19 (1) →
D	WR RS →	MA 24 (1) —◆	MB 2 (1) →	MB 2Ø (1) →
E	PAR (1) →	MA 25 (1) —◆	MB 3 (1) →	MB 21 (1) →
F	RQ CYCLE —◆	MA 26 (1) —◆	MB 4 (1) →	MB 22 (1) →
H	SPARE	MA 27 (1) —◆	MB 5 (1) →	MB 23 (1) →
J	GND	GND	GND	GND
K	MA 18 (1) —◇	MA 28 (1) —◆	MB 6 (1) →	MB 24 (1) →
L	MA 18 (Ø) —◇	MA 29 (1) —◆	MB 7 (1) →	MB 25 (1) →
M	MA 19 (1) —◇	MA 3Ø (1) —◆	MB 8 (1) →	MB 26 (1) →
N	MA 19 (Ø) —◇	MA 31 (1) —◆	MB 9 (1) →	MB 27 (1) →
P	MA 2Ø (1) —◇	MA 32 (1) —◆	MB 1Ø (1) →	MB 28 (1) →
R	MA 2Ø (Ø) —◇	MA 33 (1) —◆	MB 11 (1) →	MB 29 (1) →
S	GND	GND	GND	GND
T	MA 21 (1) —◇	MA 34 (1) —◆	MB 12 (1) →	MB 30 (1) →
U	MA 21 (Ø) —◇	MA 35 (1) —◆	MB 13 (1) →	MB 31 (1) →
V	MA 22 (1) —◇	MC RD RQ —◆	MB 14 (1) →	MB 32 (1) →
W	MA 22 (Ø) —◇	MC WR RQ —◆	MB 15 (1) →	MB 33 (1) →
X	MA 22- } —◇	SPARE —◇	MB 16 (1) →	MB 34 (1) →
Y	31=Ø } —◆	SPARE	MB 17 (1) →	MB 35 (1) →
Z	GND	GND	GND	GND
	(SOURCE 2L5)	(SOURCE 2L2Ø)	(SOURCE 2E25)	(SOURCE 2J25)

Special





October 19, 1965

To: Fabri-Tek People

Gary A. Anderson
1019 E. Excelsior Blvd.
Hopkins, Minn.

on page 7 of memo entitled

PDP-6 MEMORY BUSS SYSTEM by A. Kotok
May 28, 1965

UNDER WR RS CABLE 1, PIN #D
(where appropriate)

change:

.... All references to Split Cycle to Write or Split Cycle.

A WR RS (write restart) pulse is required by the memory control whenever a split-cycle or write only cycle is being executed; that is whenever WR RQ (write request) is asserted. In split cycle or write only cycles, the write (memory rewrite) will not start until WR RS occurs. WR RS can be sent from the processor as soon as ADS ACK is received by the processor, or any time thereafter. However, if WR RS is sent earlier than the completion of the read portion of the memory cycle, that is, earlier than approximately 1 microsecond after the ADS ACK pulse is received, the write portion of the cycle will begin immediately upon completion of the read portion of the cycle. IF RD RQ and WR RQ are both asserted, or WR RQ alone, the memory timing chain will stop upon completion of the read portion of the cycle providing WR RS has not yet been received. Then the write portion of the cycle will begin immediately upon the occurence of the WR RS signal. (In particular, the processor will have entered the memory data, MB 0-35, as pulses on the memory bus concurrently with the WR RS signal). The WR RS pulse should not be sent to the Memory Control at all during read only (RD RQ only asserted) cycles.

J. Holloway

October 19, 1965

To: Fabritek People

Gary A. Anderson
1019 E. Excelsior Blvd.
Hopkins, Minn.

on page #7 of memo titled

PDP-6 MEMORY BUSS SYSTEM by A. Kotok May 28, 1965

UNDER WR RS CABLE I, PIN #D
(where appropriate)
change:
.... ~~Whenever a split cycle or write only is requested (in general change all references to read as split cycle or write only cycle)~~

CHANGE ALL REFERENCES TO SPLIT CYCLE TO WRITE OR SPLIT CYCLE.

A WR RS (write restart) pulse is required by the memory control whenever a split-cycle or write only cycle is being executed; that is whenever WR RQ (write request) is asserted. In split cycle or write only cycles, the write (memory rewrite) will not start until WR RS occurs. WR RS can be sent from the processor as soon as ADS ACK is received by the processor, or any time thereafter. However, if WR RS is sent earlier than the completion of the read portion of the memory cycle, that is, earlier than approximately 1 microsecond after the ADS ACK pulse is received, the write portion of the cycle will begin immediately upon completion of the read portion of the cycle. If RD RQ and WR RQ are both asserted, or WR RQ alone, the memory timing chain will stop upon completion of the read portion of the cycle providing WR RS has not yet been received. Then the write portion of the cycle will begin immediately upon the occurence of the WR RS signal. (in particular, the processor will have entered the memory data, MB 0-35, as pulses on the memory bus concurrently with the WR RS signal). The WR RS pulse should not be sent to the Memory Control at all during read only (RD RQ only asserted) cycles.

J. Holloway

1 1 UNMARKED~~O~~FREE

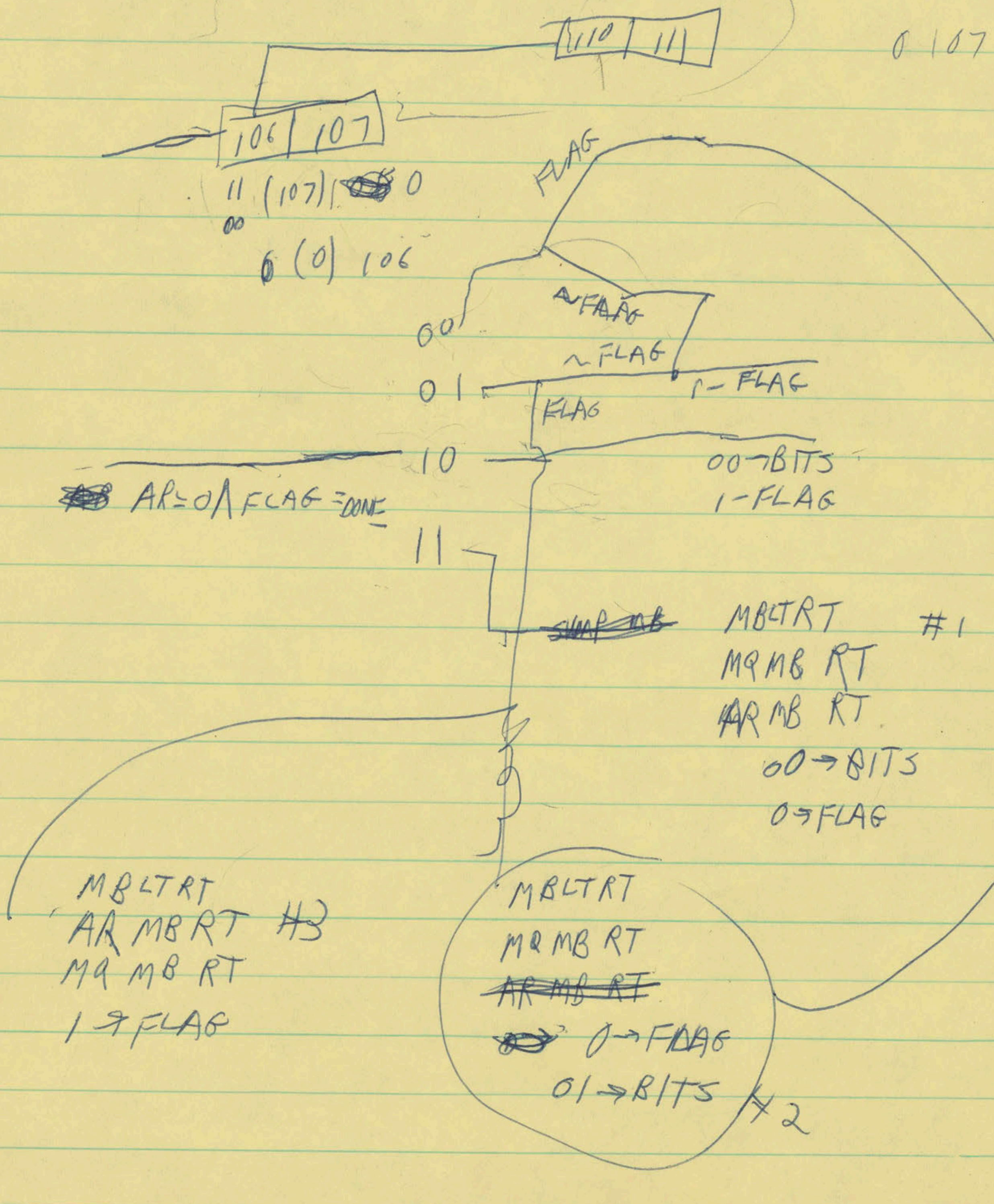
0 1 MARKED ″

00 HALF-MARKED - MARKED FULL

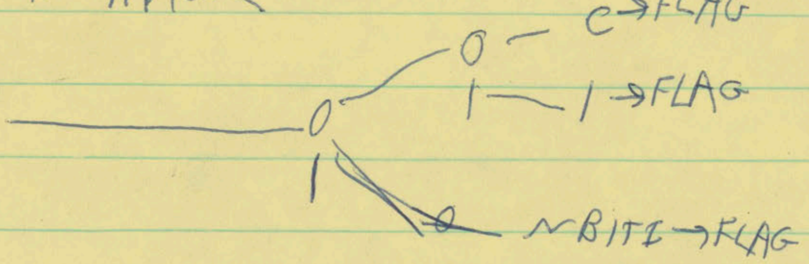
1 0 UNMARKED FULL

AR=0	105	105
MQ=105	106	107

0 107

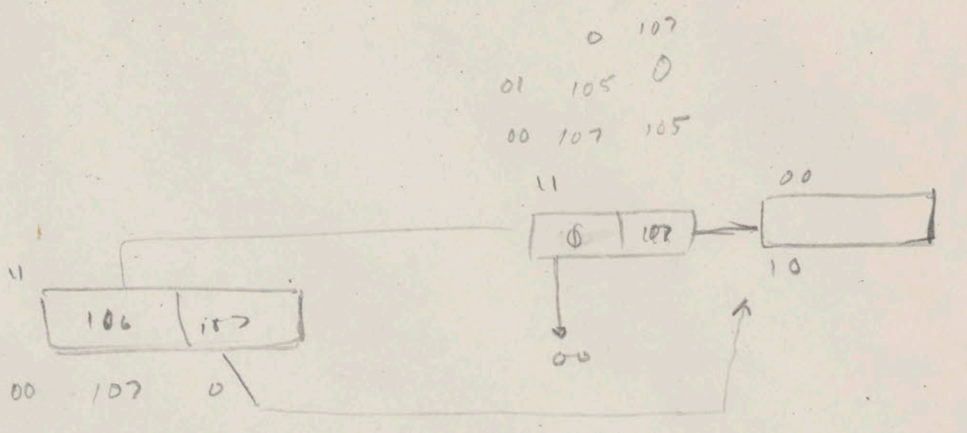


~~1→FLAG~~	~~10V(0X∧~FLAG)~~
01→BITS	00∧FLAG
00→BITS	1X
ARMBRT ~~T~~2	01∧FLAG
ARMBRT ~~T~~4	11
MBLTRTETC	11VFLAG
~~0→FLAG~~	~~11V(00~~~



INPUT language

WIDGETRY DESIGN
MOTH memory
LPT
QUAKER INS



AR	0	105	106	106	105
MQ	105	106	0	107	106

MBLT ← (ϕ)
MB RT ← (ϕ)

MB LT(J) ↔ MB RT(J)

MBLT ← AR(1)
MB RT ← AR(1)
MB LT ← AR(ϕ)
MB RT ← AR(ϕ)

MB ← PC(1)
MB LT ← MQ(1)
MB RT ← MQ(1)

MB LT ← MQ(ϕ)
MB RT ← MQ(ϕ)

AR LT ← (ϕ)
AR RT ← (ϕ)

AR LT COM
AR RT COM
AR LT ← MB(1)
AR RT ← MB(1)
AR LT ← MB(ϕ)
AR RT ← MB(ϕ)
AR LT ← MB(∀)
AR RT ← MB(∀)

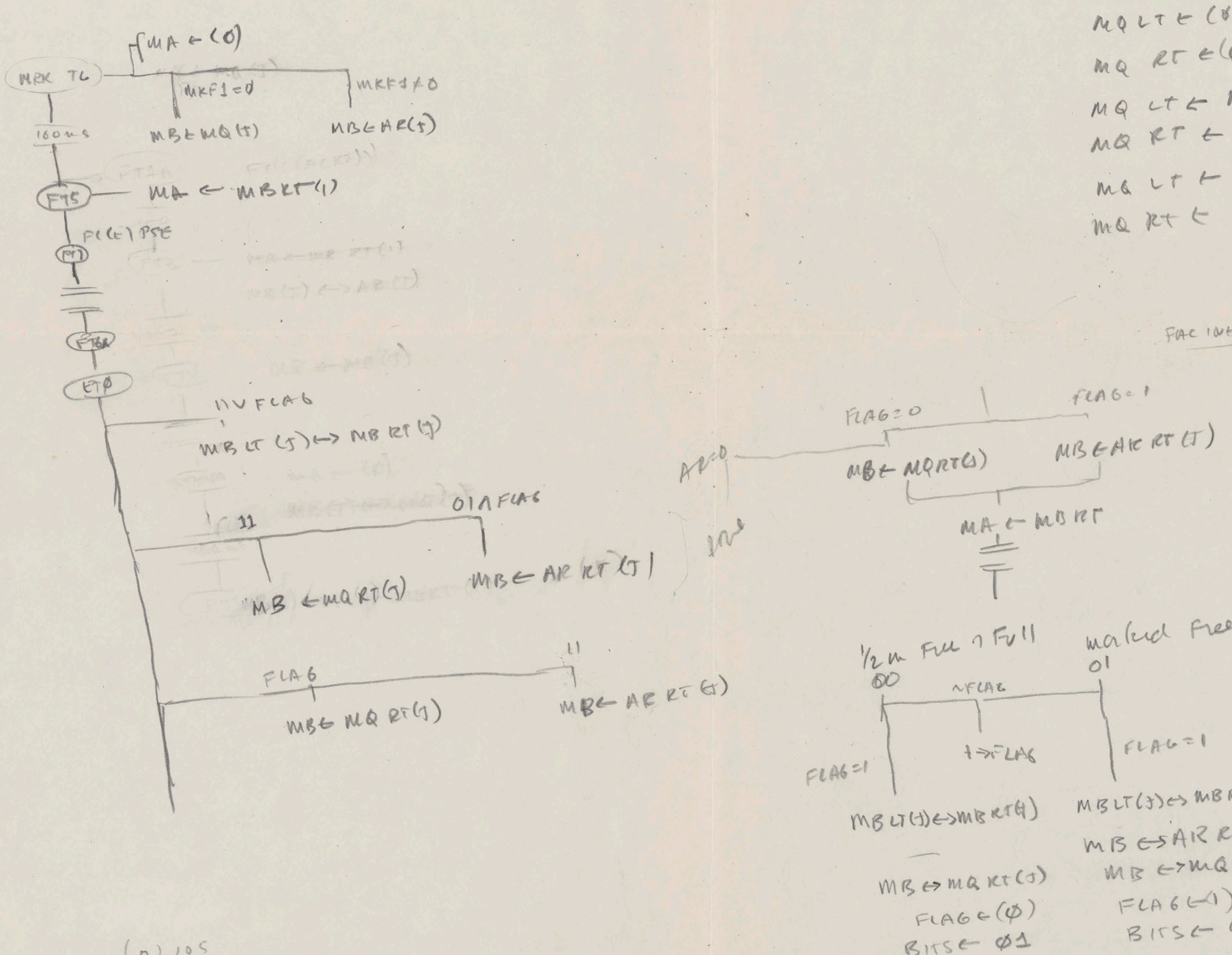
MQ LT ← (ϕ)
MQ RT ← (ϕ)

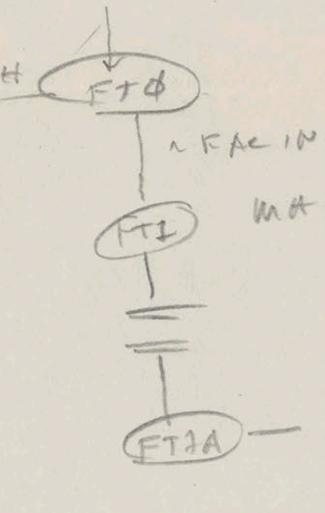
MQ LT ← MB(1)
MQ RT ← MB(1)

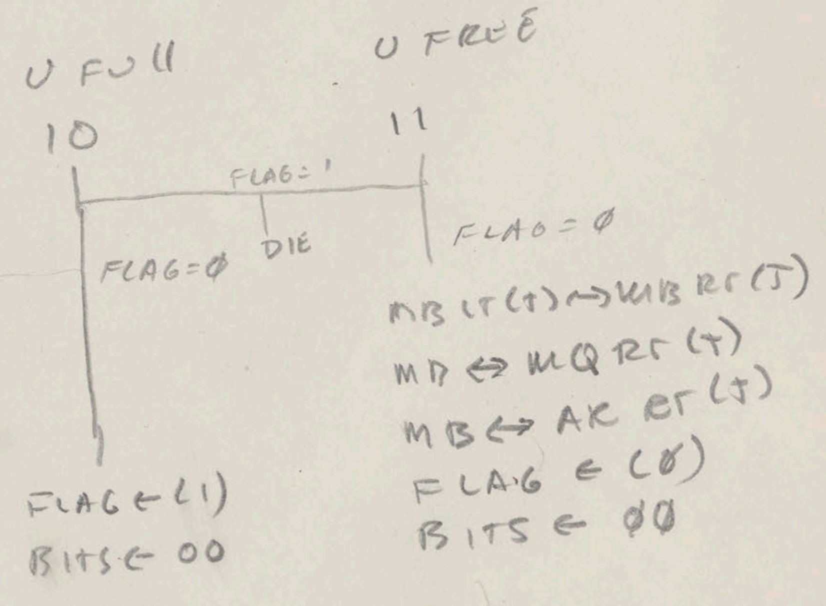
MQ LT ← MB(ϕ)
MQ RT ← MB(ϕ)

MA ← (ϕ)
MA ← MB RT(1)
MA ← IR 14-17(1)
MA ← IR 9-12(1)

MA ← PC(1)



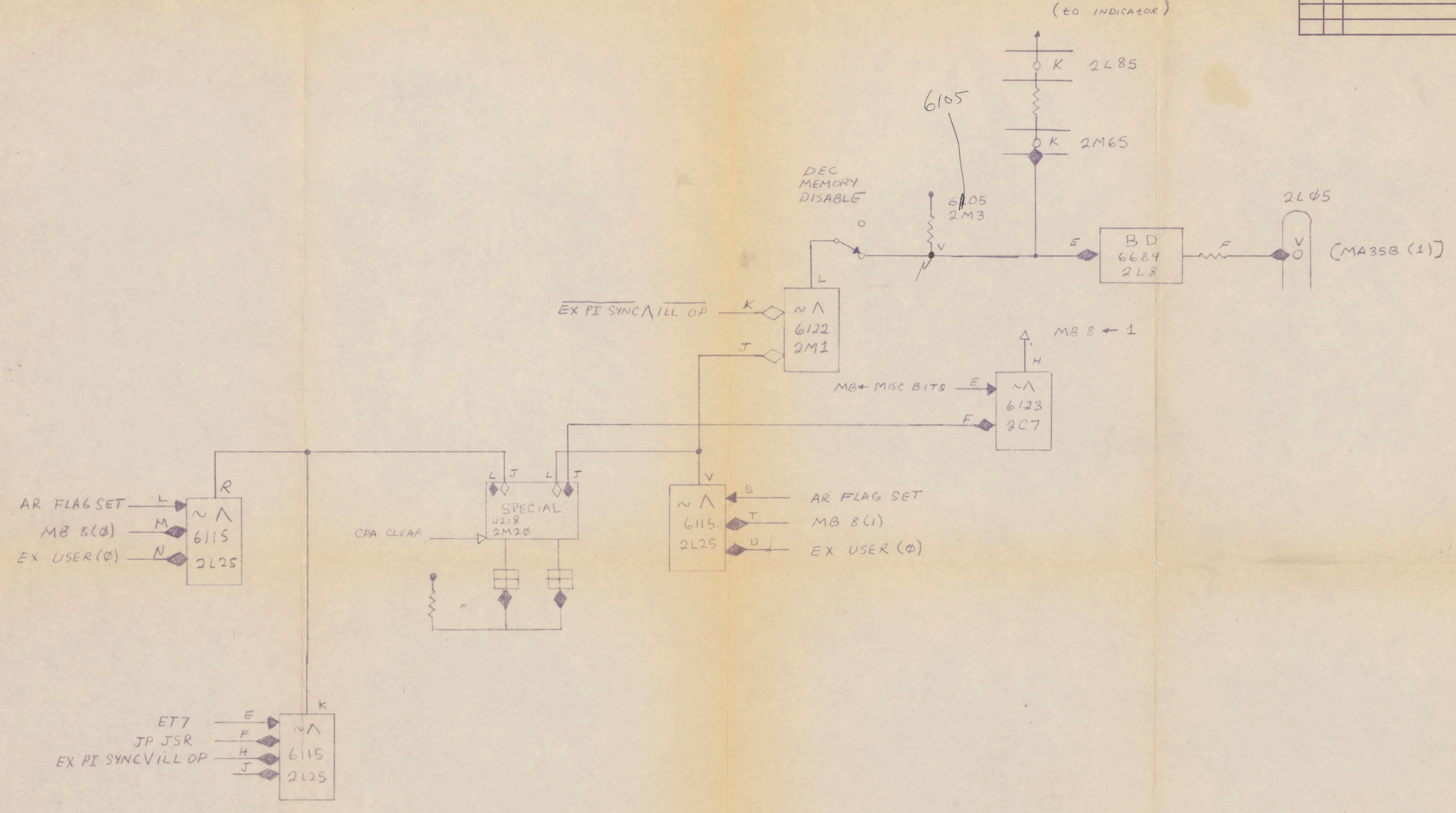




(0) 105
(105) 106

2/2 Crocks 1965-1972

DATE	SYM	REVISION RECORD	AUTH.	DR.	CK.



TOLERANCES (EXCEPT AS NOTED)	SPECIAL MODE		
DECIMAL ±		SCALE	DRAWN BY APPROVED BY
FRACTIONAL ±	TITLE		
ANGULAR ±	DATE	DRAWING NUMBER	

Notes on Displays

by

Jack Holloway

This is a semi-technical description of one possible display system for AI-MAC. It is an "idealized" one, and although it resembles the SAIL display system, the combination and extent of the hardware features envisioned may not be available from any particualr manufacturer.

Since no strident and uncompromising claims are intended here, it is hoped that the reader can temporarily put aside his own prejudices as to the relative merit of features, his urge to chop and chisel, and participate in this daydream. In general, these are things that I would find pleasurable or exciting to discover in someone´s proposal.

I favor a two-headed display system, consisting of several refresh vector-drawing displays (DPY´s) and a multitude of raster type displays. In my opinion, the DPY´s are a seperate and indespensible part of a display system, offering high resolution, good looking characters and vectors, simplicity in programming, and the ability to handle rapidly changing displays. Furthermore, it is important to have more than one or two such displays (say at least 6-8), because otherwise their software tends to be too expensive to develop for the benifit recieved, and programs that use them are restricted in usability. I think several DEC GT-40´s , Vector General, SC delta-1, or in-house displays would be suitable here.

The raster system would be patterned after the Stanford Data Disc system, with several video channels connected through a switch to a larger number of monitors. I think it reasonable that a majority of the video channels be character only display generators, and the others 512x512 shift registers for graphics uses. In addition, some number of analog sources can be selected at each monitor for viewing TV cameras, video synthesizers, or other sources of standard TV video. The switch would allow any set of channels to be selected to a particular monitor. Most users would probably find it sufficient to use the character-only channels but any display could access a graphic channel. It would probably be the case that the displays would be driven by one or more PDP-11 computers connected to the PDP-10´s with a direct memory-memory connection.

FEATURES

- The character channels would each consist of RAM storage of at least 86 by 42 characters. There would be a read-write memory shared among several channels that contained the bit-matrix definition of the 128 character set. There would be a simple processor reading PDP-11 memory that would fetch 16 bit words packed with two characters and store them into the desired character position in the selected channel´s character memory. The fetcher would interpret the following format characters.

null- ignored, not written in character memory.
tab- spaces filled in until next tab position.
return- spaces filled until right margin and line advanced.
line-feed- I would like to do away with this but it could be simulated (sigh).
rubout- special character (see below) In addition, the fetcher would detect overflow off the right margin and either wrap-around to the next line or truncate the line depending upon mode. This should relieve the software of the problem of reformating the text grossly.

-Left and right margin registers. Return would advance to the next line in the column specified by the left margin register. This allows the software to partition the screen into separate pieces of paper without treating the displayed text differently. Possibly a lower line limit also since if lines can wraparound it is difficult to determine the number of displayed lines.

-The eighth bit of the character can be used to specify a cursor (underline character) , or a bold character, or can be used with an expanded character definition store to allow additional special characters found in other character sets.

-Time-sharing features such as the rubout code which conditionally displays a blob when a IOT controlled flag is on, to display the user´s run state. An additional marker in the upper corner displayed on all consoles if the time-sharing system is detected down.

-Rubout with cursor bit is an analog see-thru character for overlapped analog and text images. Allows the user to build a window on a portion of the screen for synthesized or TV video.

GRAPHIC FEATURES

-In addition to the character channels, some number of shift register memories with 512 lines of 512 bits for each channel. The shift registers would be organized to allow rapid access to any particular line, and 16 bit transfers at near memory bandwidth.

-A memory driven display processor with character, vector, increment modes that writes into a 512x512 RAM memory. This memory could be copied into (from) any one of the shift register channels. This would allow arbitrary character sets on any graphic channel, plus a more usable graphics feature than the current Stanford Data-Disc.

-The ability to gang together channels to form a memory with 6 or 8 bits per point, driving a grey scale video synthesizer. This grey scale memory might also be written into by TV camera, allowing the program to access the data in the shift register without waiting for the frame time of the camera video, recieving samples without need to compensate for interlace, reduction in the memory bandwidth requirements so that samples with more bits can be read without critical high-speed channels, and possibly allowing several-frame averaging to increase signal/noise ratios.

-Four 512x512 channels can be assigned to a 1024 line monitor to provide extremely high quality graphics. This can be easily done by sampling among the four channels at four times the normal bit rate. The display processor could be arranged to normally assume 1024x1024 coordinates, rounded off when running into low resolution 512 channels. The display processor may even interpret the same command format as the refresh type displays.

-provision for driving an XGP type device from the selected output of one of the channels. This might entail slowing the scan rate while printing that page, which would garble that consoles screen temporarily.

PDP-11

A PDP-11 (or a few) would control the character fetcher, graphics processor and have what memory was needed per channel. Probably the text associated with the page-printer would be in the 11's memory and scrolling (glitching) of the pieces of paper would be handled by the 11. It is also possible that the line-editor function could be handled mostly in the 11. Other system wide features would be updating of the status line(s) at the top (bottom) of the screen, and mapping of characters into the other character sets.

One thing which I consider fairly important, is the ability to change large pieces of text at a rate consistent with the per-character interaction time of the system. Therefore, some sort of direct memory interface is needed with the PDP-10s. There are three possibilities that I see.

1) The PDP-10 addresses the PDP-11 as memory. To communicate with the display system, the 10 writes whatever text it wants displayed into PDP-11 memory and interrupts the 11. By assigning 11-type addresses to a page of a users map this allows the user to send his stuff directly to the display, while still providing protection to other parts of 11 memory. This is probably the most direct way of getting the stuff there. One complication is that 10 text is usually packed as five 7 bit characters per word, while 11 text would be two eight bit characters per word. Since the "port" into the 11 memory can be arbitrarily hairy, it could be arranged such that one PDP-10 word would be mapped into the correct character positions in 11 memory. For instance, locn. x000 would be written with the first address for characters to go, and then deposits into x001 would unpack the 5 characters into 2 1/2 PDP-11 words and increment the write address. There are other schemes, but one added problem is that on Foonly (you thought I was never going to mention that), it would require storing the display data through the cache (easy) but unfortunately isn't particularly efficient, since each write operation would probably be 20-30 Foonly cycles.

2) The PDP-11 addresses the PDP-10 as memory. This is almost as easy for the system as (1), it just puts the pointer to the display stuff in some fixed place and pokes the 11. The eleven can pick up the stuff while the 10 goes on, probably with the help of formatting kludges that it would reference as unique locations in its address space. Since the 10 has a paging box, the 11 may have to have one too (big deal). Also, this is somewhat more amicable to Foonly, since the grand and glorious P. Petit IO Kludge would handle all the unfortunate confusion of locating the data as it flies back and forth from the cache. This also has the marginal advantage of allowing the 11 to scan some data area (user status) in the 10 system for display which otherwise would require periodic service from the main processor. This scheme also satisfies those PDP-11 chauvinists that would put the PDP-10 in its place. Unfortunately, it may be simpler to add another "memory" to the 10 than it would be to add another "processor" to all the PDP-10 memories.

3) Some sort of memory-memory channel. This is a modification of (2). Its only advantage is an overlap of PDP-11 processing.

Also as I understand, this system is supposed to be congruent with a Multix display system. Here I can't imagine any complicated interface to the PDP-11's, just a high-speed data-phone like link, with the 11 shuffling the right data onto the right channel.

I would like to also go on record as being in favor of adopting keyboards compatible with the Stanford system. I think that the extra control bits are a complete win, and the extra characters only logical considering a 7 bit code - certainly much preferable to the non-descript <control>-X type characters. The bugaboo about "standard" character set doesn't impress me too much considering the variety of keyboards and character sets which have been used on those various systems. A much bigger advantage is gained by having all terminals with identical keyboards. To those that say that the "extra" features would be abused and lead to "bad" programming practices(!) I would recommend a Ken Colby style overlay to keep their little fingers off those keys.

```
^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^
^!
@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@
^!
@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@
^!
@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@
^!
@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@
^!
@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@
```

STANFORD TIME-SHARING PROJECT
Memo No. 37

November 2, 1965

RAID (alias TVDDT)

by Paul Stygar

Abstract: Rapid Illustrated Debugging of assembly language programs is achieved under a time sharing system by simulating a real-time console on a display device. A symbolic debugging aid which features location protection, single stepping and a dynamic view of core is described.

The research reported here was supported in part by the National Science Foundation (GP-3207).

a
4
Stanford Univ
Debugging
Display Device

RAID (alias TVDDT)

by Paul Stygar

Introduction:

RAID is a display-oriented real-time debugging aid fashioned from the grand tradition of the DDT (DEC Debugging Tape). It is available on the Philco display units under Stanford's THOR time sharing system for the PDP-1. RAID lives in core with the programmer's routines and uses the display buffer provided by the system to present to the programmer a panorama of information about the status quo of his program. The disadvantages inherent in this scheme will be discussed later.

The philosophy underlying the design of RAID is partly the desire to bring the programmer as close as possible to his code within the confines of a time-sharing system. This goal is achieved by simulating a real-time console. The availability and flexibility of the Philco display device facilitates matters greatly. A second component in the design philosophy is the desire to provide a compact but powerful command repertoire to facilitate the on-line interaction of a programmer with the utility routine which is showing him his code. A powerful command repertoire and a flexible framework enables the programmer to perform a variety of real-time debugging experiments. An effort was made to preserve the mnemonic or intuitive meaning attributed to the various commands by the other utility programs available with the THOR system.

The third component in the design philosophy is based on the observation that a debugging programmer pursues a zigzag pattern of investigation. He typically will start at some point in his program and examine the instruction flow from that point. The flow will branch, and he will examine and re-examine the various branches depending on his suspicions. Very often he will return to a line of investigation which was previously abandoned. What the programmer needs is a flexible scratchpad which will allow him to retrace his steps to pursue an alternate path. This is achieved by providing an array of locations which the programmer can generate in various ways. The contents of the locations in the array are displayed on the screen dynamically. Commands for shifting the focus of attention within the array give the programmer a backup ability which he may exercise at will to recover previous investigation paths.

As a result of these considerations - RAID is more useful and easier to use than most consoles. The programmer may observe the current contents of 16 locations simultaneously. As the programmer alters and single steps the instructions on the display, he observes the updated contents of the displayed core registers. In addition to single stepping the programmer may execute selected portions of code under the control of his program by specifying the beginning and ending of the execution. He will benefit from the dynamic display when the breakpoint instruction inserted in the program returns control to RAID. In a variation of this, RAID interpretively follows the instruction flow and stops to update the display when it is about to execute one of a pre-selected array of registers. This variation provides a slower but more versatile breakpoint system. The array of breakpoints may be modified by the program without destroying the fact that the breakpoints are stop signs. A second variation of this simulates location lockout. The programmer specifies an array of registers to be protected. RAID executes the program interpretively and halts when one of the specified locations has been changed. In either of these variations the display is left containing the sequence of instructions which resulted in the stop condition.

RAID's value as a self-instructional device is obvious: the novice programmer may enhance his understanding of the various computer instructions by executing them and observing the effects. The experienced programmer may occasionally revamp his understanding of a particular instruction. Generally a programmer single-stepping through his code will encounter occasions on which his image of the situation does not correspond to the actual situation. In writing code a programmer must anticipate the effects of the various instructions, and he must maintain an image of what his code does to the memory registers involved. When debugging with RAID, he recreates this anticipated image, and can then correct his thinking where necessary.

Display Organization:

The display consists of data appearing in fixed positions on the screen. Included are:

I. The current contents of 16 core locations specified by the programmer. The octal and symbolic contents appear side by side on the first 16 lines of the display in a form resembling FAP listings.

II. A pointer indicating which of the displayed locations is currently open. The open location is that location which will be affected if the programmer types a deposit command. The location will be referred to as "the current open location".

III. The contents of the AC (accumulator), IO (input-output register), PS (program status word) and F(program flags). These registers are simulated by the single step feature. That is, the contents of these registers are displayed as they will appear to the program being debugged.

IV. The expression being entered by the programmer, with a character pointer indicating the position for the next spacing character. After the command character has been typed, this expression remains on the display until the command is completed.

V. An occasional error message. When visible the error message indicates that the command just entered cannot be obeyed. This condition arises if the character is an illegal command or if the expression preceding the command contains unrecognized symbols.

When the programmer types a command to examine a location, we say that he "opens" the location. That location becomes the current open location. The command causes the contents of the location to be displayed on one of the sixteen lines of the display reserved for this purpose. If the location is already displayed on one of the lines, the pointer is moved to that line. A given location will not be displayed more than once. If the location opened by the programmer is not already on the display, the contents of the location are displayed on the next available line (modulo 16) below the current position of the pointer, and the pointer is moved to that line. By pre-positioning the pointer before typing the examine command, the programmer may display the contents of a location on the line of his choice.

Command Structure:

Communications to RAID are typed at the Philco keyboard in lower case. A typical communication would involve typing an assembly language expression followed by a command character. For example, to examine a location one would type the address for the location followed by a ":". Alphanumeric characters appear as expressions and as commands. The special black button to the left of the space bar is used to resolve amiguities: a character typed with the black button down is always interpreted as a command character. Characters such as ":", ">" and "<" are interpreted as command characters when the button is up. In the following discussion an underline (e.g., "S") will indicate characters which are not interpreted as commands unless accompanied by the special button.

Reverse polish notation is used: when a command requires an argument, the argument is typed before the command. The command serves as a delimiter and no action is taken until the command is typed. In the following discussion the argument is often referred to as the expression preceding the command. Each command has two interpretations, depending on whether or not an expression is typed preceding the command.

When a command character is typed without an argument, the default interpretation of the command applies. There are two major types of default interpretations. The first type concerns those commands which expect an argument to be interpreted as an 18-bit quantity. If such commands are not given an argument explicitly, the contents of the current open location are taken in lieu of the explicit argument. Examples of this are the command characters ":" and "X".

The second type of default interpretation concerns those commands which interpret an explicit argument as a 12-bit quantity specifying an address. For convenience, when not given an argument such commands operate on the current open location. When an argument is typed the location corresponding to the typed address is opened and that location becomes the current open location. The remainder of the operation of the command is identical with the default operation of the command. Examples of this are the command characters "G" and "S". In the description of such commands the default operation of the command is given first.

Replication:

To avoid typing a command character repeatedly the programmer may type a replicator with the command character. A replicator is a number typed with the special button down and immediately prefixing a valid command. The replicator causes the operation of the command to be repeated the corresponding number of times. For example, a replicator typed with the single step command "S" will initiate multiple stepping. If no replicator is typed before a command, "1" is assumed. If a zero replicator is explicitly typed no operation is performed, and the expression preceding the command is ignored.

A command typed while RAID is busily performing some task will be executed immediately. For example, by typing a space the programmer may interrupt RAID when it is preoccupied with multiple stepping. The multiple stepping may be continued by typing another replicated single step command. The use of partial results allows the programmer the flexibility to perform complex debugging experiments.

Command Arguments:

The argument of a command is defined to be the string of legal characters typed before the command character. For convenience in correcting typing errors, a backspace typed without use of the black button will erase the last spacing character typed. The argument string is interpreted as an assembly language expression consisting of octal numbers and assembly language mnemonics. The syntax is a small subset of the PASS assembler available under the THOR system. In addition to the unbuttoned backspace, the following are accepted as argument constituents:

I. 0,1,2,...,7 may be used to form octal constants. An octal constant may not contain more than six octal digits, corresponding to the PDP-1 word length. Octal constants containing the digits "8" and "9" are not recognized and will result in the error message.

II. + and - play their usual roles. A space separating two symbols is an implied "+".

III. i (the indirect bit) has the value 10000 (octal).

IV. . (the "here" symbol) has the value of the address of the current open location.

V. Symbols and labels, such as the standard PDP-1 mnemonics, the THOR Input-Output mnemonics and the programmer's address symbols have their standard values. Unrecognized symbols will result in an error message when the command character is typed.

Examine and Deposit Commands:

To open (display the contents of) a location, type an expression for the address of the location, then type the command character ":". To open the location referenced by the address part of the current open location, type ":" without an argument.

To replace (overwrite) the contents of the current open location, type an expression for the new contents. The following commands will then overwrite the contents of the current open location. With or without an argument, these commands have the listed effect:

carriage return	Moves the pointer to the next line (modulo 16);
backspace	Moves the pointer to the previous line (modulo 16);
tab or space	Has no effect on the pointer;
>	Opens location .+1 (opens a greater location);
<	Opens location .-1 (opens a lesser location).

To make minor modifications to the contents of the current open location, the following button-down commands are useful:

+	The expression preceding the command is added to the contents of the current open location;
-	The expression preceding the command is subtracted from the contents of the current open location.

To display the address part of the contents of the current open location in the notation for an assembly language constant, use "[" repeatedly.

To display the contents of the current open location in decimal, type "D".

To display the contents of the current open location as Philco characters, type either a buttoned upper case shift or a buttoned lower case shift character.

To alter the contents of the AC, IO or F, type an expression for the new contents, followed by "A", "I" or "F" respectively.

To zero core, enter "OZ". When not given an argument, the "Z" command is illegal.

The Breakpoint and Single Step Mechanisms:

The single step feature allows the programmer to execute his program one instruction at a time. Multiple stepping is achieved by typing a replicator with the single step command. The instruction in the current open location is simulated and the next location in the instruction sequence is opened. The AC, IO, PS and F are simulated. An error message is given if the instruction is one of the following criminals: an illegal instruction, a one instruction loop, a halt, an infinite "xct" chain, an infinite indirect addressing chain or a "jda AC" (cf. breakpoint). An error condition encountered during mutliple stepping will terminate the process.

The breakpoint feature is useful when the programmer wishes to execute a sequence of code without single stepping through the sequence. It is useful when the programmer wishes to use RAID immediately after his program reaches a certain point in its flow. The breakpoint is actually a "jda AC" instruction which is placed at the break location only when a "G" command is entered to transfer execution control to the program (cf. hints and kinks). The "B" command indicates where the breakpoint is to be placed. When the program encounters the breakpoint, execution control is returned to RAID and the AC, IO, PS and F are saved. A breakpoint addressed by an "xct" will cause a break. The location causing the break is opened, and the display is updated. After a "G", RAID does not resume execution control until either a breakpoint is encountered or the programmer restarts RAID.

An interesting debugging tactic involving breakpoint is the following. By planting a break in the middle of a loop and typing a replicated "G" command, the programmer causes RAID to resume control, update the display, and return control to the loop a number of times corresponding to the replicator entered with the "G". If the display includes some of the core locations modified by the loop, the programmer will see the modifications displayed as the loop progresses. This tactic can be used to checkout loops and to observe suspected bit-spreaders. The programmer has the option of interrupting the process by entering a a command. He also has the option of entering the several "G" commands separately, without the replicator.

Hints and Kinks about Breakpoints:

DO NOT try to plant a breakpoint in program-modified locations. For example, the return "jmp" from a subroutine is the wrong place for a break. DO NOT break at locations which are used as data by the program. Else let there be mystery. For example, do not break after a "jsp" or a "jda" when the subroutine expects an argument sequence to follow the calling instruction. One may place a breakpoint in the middle of a chain of indirect addressing: the right thing will happen, because "jda AC" contains an indirect bit.

The Program Status Word:

The PS combines the functions of the program counter, the extend mode status and the overflow flip-flop. It may be operationally defined as what the AC would contain if "lap", "jsp" or"jda" were executed at the location preceding the current open location. Bits (6-17) of the PS contain the address of the current open location. Bit (0) records the state of the overflow flip-flop. Bit (1) records the extend mode status. When the PS is reset the location referenced by the address part is opened. The PS is reset when a breakpoint occurs, when RAID is entered and when the commands "S", "G", "Q", "R" and "E" are preceded by an expression. The PS is simulated by these commands. In particular, the programmer may single step in extend mode.

Commands:

S The instruction in the current open location is single-stepped, and the next location in the instruction sequence is opened. An expression preceding the "S" command will reset the PS and the address referenced by the PS will be opened.

X The instruction in the current open location is executed. If the "X" is preceded by an expression, the latter is executed in lieu of the instruction in the current open location. The censorship of the "S" command is bypassed, allowing the programmer the freedom to do mischief. No breakpoint is planted. This feature is useful if the programmer wishes to treat a subroutine call as a single instruction. Location .+1 is opened if the instruction does not skip. Location .+2 is opened if it does.

B   The location for the break is taken to be the address typed before the "B". When no address is typed, the location for the break is taken to be the current open location when the "B" was typed. To suspend the breakpoint, type "5000B".

G   A jump to the current open location is executed. If the current open location is the break location, the instruction therein is first single-stepped. A breakpoint (i.e., a "jda AC") is placed at the break location, and the original contents of the break location are saved, to be restored when control returns to RAID. An expression preceding the "G" command will reset the PS before performing any of the above.

Variations on Single Stepping:

P   Protect the current open location. This causes the location to remain on the display until a "K" command is given. If the command is preceded by an address, the addressed location is opened and protected.

K   If the command is preceded by an address (e.g., ".K"), the addressed location is opened and the line on which the contents are displayed is erased. If no argument is typed, this command resets the RAID page: the display is erased, giving the programmer a "clean slate". This command is especially useful before a search (cf. the "W", "N" and "E" commands).

Q   This command simulates a "G", but with multiple kinkless breakpoints. Single step quietly (without updating the display) until one of the protected locations on the display is about to be executed. The display is then updated to show the sequence of commands preceding the stop. An expression preceding the command will reset the PS.

R   This command simulates a "G" but with location lockout: Single step quietly (without updating the display) and report the status of the program when the contents of one of the protected locations changes. The display is updated when an interruption occurs at the keyboard or when a protected location is changed by the code which RAID is executing interpretively. An expression preceding the "R" will reset the PS before initiating the interpretive execution.

The Word Search and Effective Address Search Commands:

W   An expression typed before the command causes RAID to search core for a location the contents of which match the expression preceding the command. The operation of the command is completed when such a location is found and opened or when the search has been completed. The search is continued when a "W" or a replicated

"W" is typed not preceded by an expression. The programmer should type an expressionless"W" after a word search to assure himself the search was completed.

N The operation of the command is the same as that of the "W" command, except that the sense of the search is reversed: all locations not opened by a "W" are opened by an "N" command. A not-word search is continued by typing expressionless "N" commands.

E An address typed before the command will reset the PS and the corresponding location will be opened. An effective address search is then initiated: RAID will search core for a location the contents of which effectively address the location specified by the expression preceding the "E" command. The operation of the command is completed when such a location is found and opened or when the search has been completed. Indirect addressing chains are not pursued more than 100 levels. To continue an effective address search, type an "E" without an argument.

U The upper limit of operation of the "E", "W", "N" and "Z" commands is set to the address preceding the commands. If no address is typed, the upper limit is set to 5000.

L The lower limit of operation of the "E", "W", "N" and "Z" commands is set to the address preceding the commands. If no address is typed, the lower limit is set to 0.

M The mask used in word-searching is set to the expression preceding the "M" command. If no expression was typed, the mask is reset to 777777(octal). In matching the contents of a location with the search-object, the "W" and "N" commands compare only those bits specified by the mask.

The Programmer's Address Labels:

The PASS assembler available under the THOR time sharing system provides the programmer with the opportunity to apply block structure to the organization of his program. The effect of block structure on program symbols is that at a given location in the program the only symbols visible are those labels and symbols defined in blocks containing the given location. Symbols defined in other blocks are invisible. A second effect of the block structure organization of the symbol table is that the local symbols (the symbols defined in the deepest block containing a given location) are given a higher priority. The symbol table for the local block is scanned first. In the case of the same symbol defined in several blocks containing the given location, the symbol defined in the most local block is the one that is taken.

The symbol table formed by the PASS assembler is loaded from THOR internal file 3 whenever the command "T" is typed. The table is loaded into the area between the program break and the variable break. As defined by the assembler, the program break is the last location occupied by the assembled code. The variables break is the last location reserved for a variable or a constant.

The symbol table is loaded into the space not used by the program and is trimmed to fit. When the symbol table cannot be loaded completely the symbols local to the first block completed by the assembler are the first to be ignored. A block is completed by the assembler when a "begin" and the matching"end" have both been scanned. The symbol table for the program is compiled in the order in which the blocks in the program are completed by the assembler during its single left-to-right scan. The symbol table is loaded in the order opposite to the order in which it was compiled. This means that the global symbols are the first to be available during debugging, though as the space for the symbol table diminishes even the global symbol table will be trimmed.

If the programmer suspects his symbol table is too large, he should take advantage of the block structure and the expunge feature provided by the assembler. The effect of expunging the symbols contained in debugged blocks of code is that the symbols defined in the block are not included in the symbol table which is PASS'd on to RAID. As a bonus, this increases the availability during debugging of the symbols local to undebugged blocks of code.

Sundry Bits of Information:

RAID is entered and re-entered at location 5000 (octal). The AC, IO, F, overflow status and extend mode status are saved. The display buffer and the THOR delimiter table are reset completely.

Locations 7520-7777 are a one record disk buffer which is used to load the symbol table. Otherwise these locations are available to the programmer.

The locations following the RAID entry point contain the following:

Location:	Contents:	Description:
5000	jda AC	A breakpoint
5001	777777	The word-search mask
5002	0	The word-search lower limit.
5003	5000	The word-search upper limit.
5004	5000	The break location address.
5005	0	A non-zero number here indicates the capacity in number of symbols of the space not used by the program. If "0" there is no symbol lookup.
5006	0	The variables break. Locations 5005-5006 completely govern the symbol lookup.
5007	0	The program break.
5010	30	A single-step counter.
5011	0	A positive number here indicates the number of blocks of symbols not loaded by the "T" command.

Pragmatics:

The advantage of an in-house (cohabiting) debugging aid is that the programmer may exercise an option as to whether the aid is to be used. Only the programmer in question pays to use the debugging aid. He pays by allowing the debugging aid to use part of his core. In return, he obtains the services of a debugged program which specializes in working over undebugged programs.

But there are disadvantages. Some of these arise because the debugging aid runs as a user program. This is particularly true when the debugging aid is RAID and the time-sharing system is THOR! Extensive use of RAID draws one's attention to three major problem areas.

First, the user does not have full use of core. In fact, approximately half of core is occupied by the symbol table and the RAID routines. Because RAID is easily damaged by a hit-and-run bit-spreader, RAID and the programmer are rarely on easy terms.

Second, the user has no facility to examine and experimentally alter the display buffer, though it is obviously of value to do so. One cannot reasonably single-step code which references the display buffer because RAID uses the entire buffer. There is some provision in THOR for receiving typed output at an alternate output device, though there is no provision for receiving display output per se at an alternate device.

Third, the General Input Routine provided by the THOR system allows efficient use of the input buffer allotted to each user. However, inefficiency and inconvenience result when two programs share the unique input buffer, especially when both programs provide for altering the THOR delimiter table.

Summary:

As an experiment in real-time debugging techniques, RAID as described above is moderately successful. Compared with batch-process debugging techniques such as tracing and core dumping, a real-time debugging aid alleviates the burden of anticipating the flow of the program. The programmer examines core selectively and follows his code dynamically. Compared with teletype real-time debugging aids, the use of a display device allows the programmer easier access to a larger array of useful status quo information. RAID bings the programmer closer to his code through the simulation of a real-time console.

However, the RAID limitations are not minor. It is necessary to debug large programs and display-oriented programs. Perhaps it is necessary to expand the area of co-operation with the time sharing system, perhaps to the extent of incorporating the console simulation into the time-sharing concept. Alternately, RAID would function better in a middleman role between the program and the system, perhaps as master of the former and subordinate of the latter. Console simulation is here to stay!

Acknowledgements:

The acknowledgements are in chronological order:

To Professor John McCarthy, for providing the opportunity to write a display-oriented debugging aid for the THOR system.

To Dow Brian, for making the author aware of the problems to be solved in writing a display version of DDT.

To Susan Graham, for the idea paraphrased as follows: "To debug is to iterate the process of retreating to correct a mistake."

To Dave Poole, who as author of the PASS assembler provided the program symbol labels which magically appear after the "T" command.

To the unmentioned programmers and self-appointed trouble shooters who used the early versions of RAID. Their questions and comments provided a store of things to keep in mind.

To Fran Thomson, for typing three of the n drafts of this report.

To Professor Niklaus Wirth, whose constructive criticisms fused with the author's dissatisfactions to produce the very useful display protection ("P"), multiple breakpoint ("Q") and lockout ("R") features.

Earnest

STANFORD ARTIFICIAL INTELEGENCE LABORATORY SEPT,, 1969
OPERATING NOTE 58

```
RAID      X      XXX    XXXX
X   I    X X      X     X   X
X   S   X   X     X     X   X
GOOD    XXXXX     X     X   X
F Y     X   X     X     X   X
O  O    X   X     X     X   X
R   U   X   X    XXX    XXXX
```

by Phil Petit

ABSTRACT:::
Raid is an interactive debugging program which uses the displays and allows dynamic monitoring of memory locations,

This work was supported by the Advanced Research Projects Agency of the Department of Defence under contract SD-183.

128 (3)
four bit DAC

Raid is a debugging program (RAID -- kills bugs -- get it?). Raid lives in core with your program and allows you to do various things with and to your program, such as stop at selected places and examine your core image, etc. The major advantage of Raid over DDT, is that Raid uses the displays to give you a constantly updated view of selected locations in core. Consequently, however, this means that Raid cannot be used from a teletype.

The first step in using raid is having a program to debug. For most of us this is the hardest part. The second step is to get Raid and the program loaded together:

LOADING RAID::::::::

The loader will load Raid with your program if you use the /V switch. For example if the binary file for your program was called FNORP, the loader command /VFNORP$ (where $ means alt-mode) would cause the loader (on a good day) to load Raid with your program.

Additionally, if your project-programmer number is in a special list in the system, Raid will be loaded instead of DDT when you type DEBUG. Ask a system programmer how to get your project-programmer number on this list.

Once you have your program and Raid in core, you must get into Raid. This is done by typing DDT<cr> to the system. This is because the system can't tell the difference between Raid and DDT. Stupid system. When you type DDT your display screen will flash and at the top will appear the word RAID twice. The duplexing of keyboard input will move down to the bottom of the screen. If it runs off the bottom of the screen so that you cannot see it, you can get it back by typing several dozen carriage returns. Raid is now ready to accept commands. The first command you will want to type will probably be the one which points Raid at your symbol table, but we must now pause in this tale of drama at the keyboard and discourse on the format of commands.

COMMAND FORMAT::::::::::

Raid commands consist of two parts, the first of which may sometimes be absent. The first part is a value, and the second is a command character, which may or may not have control bits with it.

Values:

A value can be a single number or defined symbol (like one of your labels), or an arithmetic expression involving numbers and/or symbols, or a machine instruction in standard assembler format, etc.

Below is a list of the various value formats and what they mean. (Arithmetic is integer.)

VAL1 VAL2 sum of VAL1 and VAL2

VAL1+VAL2 sum of VAL1 and VAL2

VAL1-VAL2 obvious

VAL1*VAL2 product (multiplication) of VAL1 and VAL2

VAL1/VAL2 quotient (division) of VAL1 by VAL2

(VAL) this causes the two halves of VAL to be swapped

VAL, If not followed immediately by another comma, this causes the value to be truncated to four bits and shifted left 23 (decimal) positions. To save you some calculating, that puts it in the accumulator field.

VAL,, this causes VAL to be placed in the left half (i.e. shifted left 18 places).

@ @ has the value 2Ø,, i.e. the indirect bit

. . has the same special meaning as in the assembler i.e. the place you currently are.

In addition, the following funny things exist:

= in front of a digit string causes the string to be interpreted as decimal instead of octal.

" followed by a chr., followed by a string not containing that chr., followed by that chr., has the value of the left adjusted ascii of the first 5 or fewer characters in the string.

' is just like " except that it is sixbit.

<ctrl 1>' (this means the ' character with the control-1 key down) this causes the string of characters following it, up to the first non-letter

non-digit character to be converted to radix50, and has that value.

<ctrl 1>% followed by a string of values(not containing comma) seperated by commas, causes the values in the string after the first to be considered bytes of a size indicated by the first value in the string.

EXAMPLES::::

=10 has the value 12 (octal)

"/POT/ has the same value as ASCII /POT/ has in the assembler, namely -- 502372400000

'/DSK/ has the same value as SIXBIT /DSK/ has in the assembler, namely -- 446353000000

<ctrl 1>%3,5,4,3,7,0,7,1 has the value 543707100000

RAID COMMANDS:::::

Raid commands consist of a single character, sometimes with control bits, sometimes preceded by a value. In the following discussion, <ctrl 1> preceding a character will mean that character, typed with the control-1 key down. Similarly, <ctrl 2> will indicate the control-2 key, and <ctrl 12> will indicate both keys.

The following things should be kept in mind. Raid allows you to see the contents of various locations in core by displaying or "opening" these locations on your display screen. It places a pointer next to the most recent location opened. These locations are usually displayed in the order in which they are asked for, starting at the top of the screen and going down. One exception to this is the < command; the other is if the location already appears on the screen. In the first case, locations are displayed going up the screen; in the second case, the pointer is moved to the place on the screen where that location is already displayed. There is a maximum number of locations which can be displayed at one time. This maximum number can be set with the I command, and is initially 10 (i,e. 8). When the bottom of the screen (based on the maximum number) is reached by the pointer, it wraps around to the top, and starts replacing the locations there with the new ones. When the pointer reaches the top while moving up, it wraps around to the bottom.

The normal pointer consists of a "both-ways arrow" (↔), but there are two other kinds of pointers which are used sometimes. They are left arrow (←) and right arrow (→). The both-ways arrow is the main pointer, and, as long as there is no right arrow on the screen, the location pointed to by the both-ways arrow is the value of "dot" (i.e. .). However, if there is a right arrow on the screen, the value of "dot" will be the location pointed to by the right arrow. In the following discussion of the Raid commands, the phrase "the location pointed to" will always refer to the location whose value is "dot", i.e. the location pointed to by the right arrow, if any; else the location pointed to by the both-ways arrow. This all sounds fairly confusing, but the confusion is in the description rather than in what Raid does.

Locations are opened and displayed in various modes. Raid initially opens locations in "cymbolic" mode (symbolic?). This means that words are displayed, as much as possible, as machine instructions in standard assembler format. All this can be changed, however, with the mode commands listed below. All the mode commands use one or more control bits (they are all typed with one or more control keys), and the significance of the different combinations of control bits is the same for all of them. If a mode command is typed with only the <ctrl 1> bit, the mode of the location pointed to by the both-ways arrow is changed, and nothing else. If it is typed with <ctrl 2>, then the mode in which FUTURE locations are opened is changed, until a carriage return is typed, at which point the mode reverts to the "standard" mode -- initially cymbolic. If it is typed with both control bits (<ctrl 12>) the mode for future locations is changed as with <ctrl 2>, but the "standard" mode is also changed. Note that with both the <ctrl 2> and the <ctrl 12>, none of the modes of locations already on the screen is changed.

C This sets the mode to "cymbolic". This means that words will be displayed as machine instructions, if possible, and that fields (address, index, etc.) will be displayed as symbols from your symbol table (see below) plus or minus an offset, if possible. See appendix A for a full discussion of this mode and the parameters which determine what it does.

O This sets the mode to "octal". Words are displayed as octal (base 8) numbers, except that a space is inserted between the left and right halves of words if the left half is not zero.

D This displays words as decimal numbers, preceded by an equal-sign (=).

F This displays words as decimal floating-point numbers, unless they are not normalized, in which case they are displayed in decimal mode (above).

H This is half-word mode. The left and right halves of the word are displayed, symbolically (i.e. as symbols from your symbol table, plus or minus an offset), separated by a double comma.

T This is ascii, or 7-bit type-out mode. The word is considered to contain 5 characters of 7 bits each, left adjusted in the word and these 5 characters, plus a possible sixth if the low order bit is on, are displayed. Carriage return, line feed, and form feed appear as a small CR, LF, and FF, respectively. If the high order 7 bits (first character) in the word is Ø (null), the word is considered to contain right adjusted ascii.

VAL T This is for other character type-out modes. Legal values for VAL are: 7 for ascii (as above), 6 for sixbit, and 5 for radix5Ø.

Q This is byte-pointer mode. The word is displayed in exactly the format used by the assembler POINT pseudo-op, that is, the word POINT followed by a size field, followed by a comma, then the address, index, etc., then possibly another comma and a position. The size and position are decimal.

VAL V This is byte mode -- the output analog of the <ctrl>% input mode. The word is typed out as an appropriate number of bytes, separated by commas. The bytes themselves are typed in octal. The VAL is the byte size and should not be negative. A byte size of Ø has a special meaning which is described in appendix A.

A This is absolute mode. This one is different from all the others in that it does not change the basic mode of the mode being changed, but merely indicates that, while the word is being typed out in the same mode as before (the last mode indicated for it), addresses and other fields are typed as numbers instead of symbols.

U This is the same as cymbolic mode except that the address field (right half of the word) is considered to be up to three right-adjusted ascii characters and is typed that way.

VAL U This is as above except that VAL indicates: if it is 5, then radix50 for the address field, if it is 6, then sixbit, and if it is 7, then ascii, as above.

The following is a listing of the commands to examine and change core.

VAL <ctrl 1> : This points Raid at the symbol table for the program indicated by VAL. In this case, VAL should be a single identifier.

VAL <ctrl 1> & This points Raid at the symbols (within the program it is already pointing at) for the block indicated by VAL. Again, VAL should be a single identifier. If you don't use block structure, ignore this command.

VAL ; This causes the location VAL to be displayed. If it is already on the screen, Raid just moves the pointer to that position, otherwise, it displays it in the next location on the screen.

VAL <ctrl 1> ; This causes the location VAL to be displayed as above, except that the location is also protected. This means that a star appears on the screen to the left of the location and that location cannot be erased from the screen.

<ctrl 1> ; This, which is the same command as above, but without the VAL, causes the location Raid is currently pointing to to be protected.

<ctrl 2> ; This causes the location Raid is currently pointing to to be unprotected.

> This causes Raid to display the next higher location from the one it is currently pointing at. For example, if it is currently pointing at location 36, this command would cause it to display (and point at) location 37. If the location is not already on the screen, the pointer moves down one position to display this next location.

< This causes Raid to display the next lower location from the one it is currently pointing

at. For example, if it is currently pointing at location 36, this command would cause it to display location 35. If the location is not already on the screen, the pointer moves UP one position to display this next location.

<ctrl 1> > This causes Raid to move its pointer down one position without changing what location is displayed there. As always, if the pointer is at the bottom, moving down causes it wraps around to the top.

<ctrl 1> < This causes Raid to move its pointer up one position. As always, if it is at the top, moving up causes it to wrap around to the bottom.

<ctrl 2> > This is the same as > without any control, except that instead of displaying the new location in the current mode, it displays it in the same mode as the location it is currently pointing to. For example, if you open the first word of your teletype buffer, and change the mode for it to ascii, you can then open the second word of the buffer in ascii by using this command.

<ctrl 2> < This is the same as < except for mode as above.

<ctrl 12> > This is the same as <ctrl 1> > except for mode as above.

<ctrl 12> < This is the same as <ctrl 1> < except for mode as above.

VAL <ctrl 1> I This causes the number of lines Raid is willing to display to be set to VAL. This number is initially 10 (i.e. 8).

<ctrl 1> I This causes the number of lines Raid is willing to display to be increased by 1.

<ctrl 2> I This causes the screen to be cleared of all displayed locations. This puts the screen back to the condition it was in when Raid was first loaded, except that it does not change the program or block pointed to.

<cr> Carriage return, not preceded by a value, has no effect except to cause the screen to be updated.

VAL <cr>

Carriage return, preceded by a value, causes that value to be placed in the location currently pointed to (by the right arrow, if any, else by the both-ways arrow). That is, the location whose value is dot (i,e. .),

VAL >

This has the same function as > except that the VAL is first placed in the location pointed to, as with carriage return. ALL forms of > and < (as well as TAB and all forms of [,], and @) act as if they were preceded by a value and carriage return, if they are preceded by a value; i.e. they deposit that value in the location currently pointed to before having their standard effect. These commands are not all listed seperately.

<TAB>

Tab causes the location whose address is in the right half of the word currently pointed to by the both-ways arrow to be displayed. Note that if TAB is preceded by a value, that value is first placed in the location pointed to, and THEN the location indicated by the right half OF THAT VALUE is opened.

;

A semicolon, with no control bits and not preceded by a value, causes the location indicated by the right half of the word currently pointed to by the both-ways arrow to be opened, as with TAB, except that, while the both-ways arrow moves to point to the new location, a right arrow is left behind at the old location, except if there is already a right arrow on the screen, in which case, the right arrow stays where it is. This means that the value of dot is not changed, and that if something is deposited, it is deposited in the old location (where the right arrow is). Note, however, that if a further semicolon is typed (by itself), the location opened will be the one indicated by the right half of the new word, the word pointed to by the both-ways arrow, Note also that typing this command preceded by a value converts it to a different command; see above.

<ctrl 1> [

This has the same effect that semicolon by itself has, namely it opens the location indicated by the right half of the word pointed to by the both-ways arrow, but does not change dot; except that it can be preceded by a value, in which case that value is deposited in the location currently pointed to (by the right arrow, if any -- i.e. dot) BEFORE opening the location indicated.

<ctrl 2> [This has exactly the same effect as TAB. Note that preceding it with a value causes that value to be deposited BEFORE opening the indicated location.

<ctrl 1>] This has the same effect as <ctrl 1> [except that it uses the left half, instead of the right half. Again, it leaves a right arrow behind, unless there is one already, and it may be preceded by a value, which will be deposited BEFORE opening the indicated location.

<ctrl 2>] This has the same effect as <ctrl 2> [except that it uses the left half. This one can be preceded by a value too. If you think you're getting tired of reading all this detail, think how tired I am of typing it.

<ctrl 1> @ This has the same effect as <ctrl 1> [except that it uses the EFFECTIVE ADDRESS of the location currently pointed to by the both-ways arrow. Yet again it leaves behind a right arrow (if none yet) and may be preceded by a value. You can figure out what happens if it's preceded by a value, can't you?

<ctrl 2> @ I bet you can figure this one out by yourself. That's right, it has the same effect as <ctrl 2> [except it uses the effective address instead of the right half.

<ctrl 12> [This and the next two commands are a little complicated. This causes the location currently pointed to by the both-ways arrow to be protected (remember the little star that keeps things on the screen?), then opens and protects the location addressed by the right half of the location pointed to by the both-ways arrow, but does so DYNAMICALLY. This means that whenever the right half of the word pointed to by the both-ways arrow changes (that is, the word which was pointed to by the both-ways arrow at the time this command was given), the location which was caused to be displayed by this command is changed to be the location NOW indicated by the right

half of the word which was pointed to by the both-ways arrow at the time the command was given. Is that clear? Perhaps an example would help. Suppose you wanted to keep track of the top location of the stack. You would say P; , which would cause the push-down pointer (which everyone but DEC calls P) to be displayed (and pointed to). Then you would say <ctrl 12>[. This would cause the P location to be protected, then would open the location addressed by the right half of P, and cause it to be protected. However, from then on, whenever P changed, the location you just displayed would change to be the location addressed by the right half of P -- or the top location of the stack.

A few special things are true of the location opened by this command. First of all, you should avoid pointing to this location on the screen and trying to deposit in it. You will wind up depositing in a table inside Raid, and your screen will do funny things. You are helped in this by the fact that this command leaves a right arrow behind, so the only way of pointing to this funny location is with the <ctrl 1> > command and its friends. If you open this same location by normal means, it will be opened in another place on the screen, and you can deposit in it there. For example, if, in the P example above, the right half of P addressed location 300, and you said 300; , you would get location 300 opened in two places on the screen. The old one would be the dynamic one, and the new one would be an ordinary one.

It is possible to "chain" this command with itself (or with the next two, in any order or combination), for as many locations as you have room on the screen. For example, if, in the P example we have been belaboring, you had said <ctrl 12>[twice, instead of once, you would have opened the location addressed by the right half of the word on the top of the stack. This is all dynamic -- now to two levels -- so that if P changes, you have displayed, not only the new thing that P points to, but also the new thing that the new top of the stack points to.

Note, finally, that if this command, or either of the next two, is preceded by a value, that value is deposited in the location currently pointed to (by the right arrow, if any), as above, BEFORE the operation takes place.

<ctrl 12>] This is exactly the same as the <ctrl 12>[command above, in all its confusing glory, except that it uses the LEFT HALF instead of the right half. Will wonders never cease? No.

<ctrl 12> @ This is exactly the same as the <ctrl 12>[command above, in all its confusing glory, except that it uses the EFFECTIVE ADDRESS instead of the right half. Will wonders never cease? Yes.

VAL <ctrl 1> = This causes a word inside Raid to have VAL (not the contents of VAL, but VAL itself) deposited into it, and that location to be displayed on the screen. The location is opened in octal mode, but the mode can be changed, as with any other location displayed. This cammand leaves a right arrow behind, if there isn't already one. This, then, is a way of seeing what some value is in other modes. For example, you might want to see what the octal value of the label FOO is (say FOO<ctrl 1>=), or what the cymbolic representation of 346 might be.

VAL : VAL should be a single identifier. This command causes a symbol, or label, with the name of whatever VAL is, to be created and put into your symbol table. The symbol is put into the block you are currently inside, and is given the value of the current value of dot, i.e. the location currently pointed to.

VAL ← This command differs in format from the standard. It should be followed by another value, VAL1. VAL should be a single identifier, but VAL1 may be any value. This command creates a symbol with the name VAL (as does the : command), but sets its value to VAL1. VAL1 should then be followed by a carriage return.

VAL <ctrl 1> K This causes the symbol VAL (which should be a single identifier) to have a bit turned on in its symbol table entry which causes Raid to not use the symbol for typing out the contents of locations. In other words, this has the same effect as using double left arrow (←←) in the assembler.

VAL <ctrl 1> W This is the word-search command. The general effect is to find all words which have VAL in them. Specifically, Raid searches core, between certain limits, which may be set (see below), for words which match VAL (not the contents of VAL, but VAL itself) in all bit positions which are on in location $M. That is to say, two words are compared by XORing them, and then ANDing the result with location $M. If this produces Ø, the words to match. Raid continues to search, opening each location which matches, until it comes to the end of the range, or until it has found enough matches to half-fill the display locations available, at which time it stops and prints a big star (*) on your screen. You may, at this point, type ∨ (the "or" key) to continue the search, and Raid will pick up where it left off, stopping when it has again half-filled the screen; or you may type any other command. (No characters are lost.) However, if you do not let a search run to completion, the next search you do will take up where the last one left off. You can, at any time later, type ∨ and continue the last search you did.

VAL <ctrl 1> N This is not-word-search. This works exactly like word-search, except that words are considered to match only if they are different in some bit which is on in $M.

VAL <ctrl 1> E This is effective address search. It works like word-search except that for each word examined, the effective address is calculated, and this effective address is matched with VAL. The mask in $M is not consulted, and words are considered to match if VAL and the effective address are exactly the same.

∨ (The "or" key). This causes the last search you did to be continued. If you have done no searches, or if the last search you did has already run to completion, this command does nothing.

VAL ∪ This causes the lower search bound, for the next search only, to be set to VAL. At the completion of the next search, this bound will be set back to its original value.

VAL <ctrl 1> ∪ This causes the lower search bound to be set permanently to VAL. This sets the value back to which the search bound will be set at the completion of a search.

VAL ∩ This sets the upper search bound, for the next search only.

VAL <ctrl 1> ∩ This sets the upper search bound permanently.

The following section describes the Raid commands which allow you to run your program in various ways. These include the commands for manipulating breakpoints, which cause your program to pause when it gets to selected places, so that you can poke at it and see what's wrong. There are also single step features which allow you to run you program one instruction at a time, while displaying important locations.

Associated with several commands (including the searches above), is a big star (*), which Raid prints on your screen to let you know it is done with something which may have taken it a while. This star, in all cases, is removed the next time Raid recieves input -- usually as soon as you type anything.

VAL <ctrl 1> G This is the go command. It causes Raid to start running your program at location VAL. Raid actually transfers to your program, so your program will continue to run until it hits a breakpoint, if any, or you say control-C, or your program exits or does something the system doesn't like.

<ctrl 1> G The go command, without a value, starts your program at its starting address, i.e, the address in the right half of JOBSA.

VAL <ctrl 1> B This causes Raid to plant a breakpoint at location VAL in your program. What Raid actually does, is change this location to a JSR to a certain location in Raid, remembering what the location used to be. This means that when your program gets here, it will JSR to Raid, at which point Raid will put the location back to what it was, open the break-point location on the screen and print a big star on the screen. You are now in Raid and can type commands to it. If you examine locations, or move the both-ways arrow in any other way, Raid leaves behind it a left arrow. Raid always prints a left arrow at the next location your program should execute, if that location is on the screen, and if the both-ways arrow is not there.

VAL <ctrl 2> B This removes the break-point, if any, at location VAL.

<ctrl 2> B This removes all break-points.

<ctrl 1> P This causes Raid to continue running your program from where it left off. If your program hit a breakpoint, Raid will continue your program with the breakpoint instruction (executing the real instruction there), and your program will run until it hits another (or the same) break-point, etc. If you have been stepping your program (see below), Raid starts it up at the next location to be executed.

VAL <ctrl 1> P This causes Raid to procede (as above) from the current break-point (the last one you hit), VAL times. That is, it has the effect of saying <ctrl 1> P, VAL times, as long as you hit the same preakpoint each time. If you hit other breakpoints in between, you will stop, then if you procede from them and hit the old breakpoint, the counting will continue. See appendix B for details of getting out of this, etc. This description is not, I realize, totally unambiguous. When a description is ambiguous, keep in mind that Raid generally does the right thing.

<ctrl 1> ← This causes Raid to open, on the screen, the next location your program is going to execute, i.e. the next step location, or the last break-point you hit.

<ctrl 1> S This is the basic step command. It causes the next location in your program (the next location to be executed; the left arrow location) to be stepped. This means that the instruction has its effect, and then you are back in Raid. It is as if you had planted a break-point on the next instruction you would get to, and proceded. After stepping, Raid opens the next location (the next one you will execute). It does not print a star. If the instruction you step skips one instruction, Raid also displays the instruction skipped.

<ctrl 2> S This is exactly like <ctrl 1> S except that instead of stepping the next instruction of your program, it steps the instruction currently pointed to (by the RIGHT arrow, if any, else by the both-ways arrow -- i.e. dot). It then opens the location that that gets you to, and it becomes the next location to be executed. Note that this is a way of getting started with stepping, if you haven't run any of your program yet, or if you want to change the flow of your program.

<ctrl 1> X This is the basic execute instruction. It has the same effect as <ctrl 1> S, except if the instruction to be stepped (executed), is a subroutine call instruction (JSR, PUSHJ, JSA, or JSP), or a user UUO. In these cases, it treats the instruction, and the subroutine (or UUO

routine) it calls, as one instruction. This means that your program starts running at the subroutine call (or UUO), and runs until it returns, and stops on the instruction it returns to. This instruction is then opened. Note that if you STEP a user UUO, you wind up inside your UUO routines. There is a restriction involved in this command, and it applys also to the next two commands (the other two X commands). The restriction concerns how many locations a subroutine (or UUO) may skip. The maximum is 7. If you execute a subroutine call, or a UUO, (currently no system UUO skips more than 1, except INIT, which is handled as a special case by Raid.), and it skips more than 7 locations, you will wind up in a funny place in Raid and all sorts of wrong things will happen. A few words should be said about break-points in executed subroutines. In general, they work. You may hit a break-point inside a subroutine, the call to which was executed, and you may then step and execute other instructions. When you procede from the breakpoint, you will get back when the subroutine exits, just as if you hadn't hit any breakpoints. You should NOT step the subroutine return, as you will wind up stepping locations inside Raid. If you do this accidently, and haven't gone too far, you may procede (<ctrl 1>P) and the right thing will happen. You may nest executes to a level of 8. You should avoid executing subroutine calls which you don't return from, as you will remain inside the subroutine, as far as Raid is concerned, until you do return, and this will decrease the number of levels you can nest subroutines.

<ctrl 2> X This works just like <ctrl 1>X except that it starts with the instruction currently pointed to.

VAL <ctrl 12> X This causes the instruction VAL (VAL itself, not the instruction at VAL) to be executed as though it was in your program. Executing the instruction has no effect on which instruction is the next instruction to be executed, even if VAL is a jump or skip. VAL may be a subroutine call, in which case the right things happen. The restrictions listed above for executing subroutine calls apply. Note that executing a JRST with this command has no effect (except possibly on flags).

VAL <ctrl 12> Y This has the same effect as <ctrl 12>X if VAL is a subroutine call. Otherwise, the instruction is just plain (vanilla) executed, regardless of what it is. Even jumps are not interpreted. If the instruction does not jump, it should not skip more than two, and control reverts to Raid as with <ctrl 12>X. If the instruction does jump, you are off and running, as with <ctrl 1>G, until you hit a breakpoint or something. The purpose of this instruction is to augment the <ctrl 1>G command. It is to be used with things like flag-restoring JRSTs, and such, and has only minimum utility, except in the exec Raid version.

<ctrl 12> S This is the multi-step command. It has, except as noted below, the same effect as repeatedly saying <ctrl 1>S. It steps the current location, updates the screen (displaying the next instruction to be executed), steps the next instruction, updates the screen, and so on. It keeps running until either you type a key, in which case it stops and returns control to Raid (the character you type may be anything, and is ignored); or it gets to a subroutine call or subroutine return instruction (Subroutine return instructions are: POPJ, JRA, and JRST @). When it reaches one of these, it pauses and displays a big star. If you type S, and it is a subroutine call, it steps the call and continues with the multi-stepping. If you type X and it is a subroutine call, it executes the subroutine call and continues with the multi-stepping. If you type S or X and it is a subroutine return, it continues with the multi-stepping. If you type anything else (except as noted in appendix B), for instance space, it returns control to Raid, and you must type <ctrl 12>S to restart multi-stepping. UUOs are treated as subroutine calls.

This ends the discussion of Raid commands. You now know as much as (more than?) you need to to use Raid. It is suggested that at this point you go poke at Raid a little to become famillīar with it. If you want, you may read the appendices at this time, but it is probably better to wait until you have used Raid and are more famillIar with it,

APPENDIX A

CYMBOLIC MODE:::::::::::::::::::::

This describes how cymbolic mode decides how to display a word. If the left half of the word is Ø, it is displayed in halfword mode. If the left half is all ones, the word is printed as a negative number. If the left-hand nine bits (opcode) is Ø or 777, the word is printed in halfword mode. Otherwise, if there is an entry for the opcode, the word is printed as an instruction. If there is no opcode entry, an opdef entry is searched for, and if none is found, the word is printed in half-word mode.

If the word is printed as an instruction, the index and accumulator fields are printed as symbols only if there is an exact match, otherwise as numbers.

PRINTING SYMBOLS:::::::::::::::::

When Raid is going to print a number, unless it is in absolute mode, or some mode where numbers are always printed as numbers, it tries to print the number as a symbol, plus or minus an offset. To do this, Raid first searches the symbol table for the two best matches with the number it has, one greater, the other less. If the number is less that 14Ø, Raid requires an exact match, or it prints it as a number. Otherwise, if it has found an exact match, it prints that symbol. If not, it goes through some contortions to decide which close match to use, and whether or not it will use either. There are four parameters it uses in deciding. These parameters are stored at location $C ff. The first parameter, the one at $C, is the maximum plus offset. The second one, at $C+1, is the maximum minus offset. Both of these numbers start out at 77, but may be changed. The value is anded with 777 before use. The third parameter we will call S, and is initially 1Ø, the fourth parameter we will call Q, and is initially 4Ø. We will call the plus offset P, and the minus offset M. Raid first compares P and M with their respective maxima. If both are too big, the number is printed as a number. If P is too big but M is not, then M is used (the minus one). If M is too large, but P is not, then P is used. If both are within the limits, then the fuction F=((P*Q)/1ØØ)-S-M is calculated, where 1ØØ is octal. If F is positive, M is used, otherwise, P is used. This means that S and Q are relative weighting parameters; S is additive weighting, and Q is multiplicative. Notice that if Q=1ØØ and S=Ø, Raid uses the smaller of M and P. If Q is instead 4Ø, P is used unless it is twice as big as M. On the other hand, if Q=10Ø, but S=1Ø, P is used unless it is greater that M by more than 1Ø.

BYTE SIZE Ø:::::::::::::

If the byte size in the V mode command, or in the <ctrl 1>% input string is Ø, the bytes are interpreted acording to a mask in location $M+1. Bytes may be any sizes, and the boundaries are indicated by a change from Ø's to 1's, or vice-versa, in this mask. For example, if $M+1 contains 7Ø7Ø7Ø7Ø7Ø7Ø, this would indicate 3-bit bytes. 77ØØ77ØØ77ØØ would indicate 6-bit bytes. 7417Ø36Ø7417 would indicate 4-bit bytes. 77Ø77Ø77Ø77Ø would indicate a 6-bit byte, followed by a 3-bit byte, followed by a 6-bit byte, followed by a 3-bit byte, etc. 252525ØØØØØØ would indicate 18 one-bit bytes followed by a 18-bit byte.

RAID DEFINED LOCATIONS::::::::::::::::::

RAID This is the starting address of Raid.

DDT This is the same as RAID.

DDTEND This is the first unused location after Raid. It is not very useful now that the loder moves the symbol table down (up?) to just beyond the last program loaded.

$C This and the three locations following it are the parameters for deciding how to print symbols. See above.

$M This location is the search mask. It is initially set to -1. See the search commands.

$M+1 This is the byte mask for Ø byte size. See above.

$I This is the location where Raid keeps its current idea of your program counter -- the address of the next instruction to be executed. Breakpoints JSR to this location. The left half contains your program flags. If you change the left half of this word, you will change what program flags get restored each time your program is started up.

$1B to $20B These 20 locations, and the four locations following each of them, are the breakpoint table. For a detailed description of what each contains, see appendix B. The first word is the address of the breakpoint. The location is -1 if this breakpoint is unused. The second location is the multiple procede count. The third location is the conditional skip instruction. The fourth location is string-breakpoint byte pointer. The fifth location is the real contents of the breakpoint location.

APPENDIX B

BREAKPOINTS:::::::::::

You may have a maximum of 2Ø (octal) breakpoints at any one time. The breakpoint information is contained in a table, with five locations for each breakpoint. The first location of each of the 2Ø enties is given a label of the form $nB, where n is a number from 1 to 2Ø. This first location of each entry contains the address of the breakpoint in your core. It is -1 if this breakpoint is unused. Entries are assigned to breakpoints in the order in which breakpoints are created, starting with $1B. The real contents of the breakpoint location are stored in the fifth of the locations in the table entry (i.e. $nB+4). Changing this location, however, has no effect, because the real contents are replaced in your core while you are talking to Raid, and the JSR is placed there only while your program is running. The second location in each entry ($nB+1), contains the multiple procede count for that breakpoint. This is where Raid puts the count if you say VAL <ctrl 1> P. This count is counted down by 1 each time you hit this breakpoint and the breakpoint is ignored (you procede automatically) if the count is still positive. Depositing a number here will have the same effect as using multiple procede. Depositing Ø here will get you out of a multiple procede.

The third word of each entry is the skip instruction. If this instruction is non-zero when Raid hits this breakpoint, Raid executes the instruction (which may be a subroutine call), and what Raid does with the breakpoint depends on whether or not this instruction skips. If the instruction does not skip, Raid does the normal thing (what it would have done if this word had been Ø), namely, it counts the multiple procede count and procedes if it is positive, stops if it is zero or negative. If the instruction skips one, Raid does not count the multiple procede count, but rather it stops at the breakpoint anyway. If the instruction skips two, Raid does not count the multiple procede count, but rather it procedes (ignores the breakpoint), anyway. The instruction had damn well better not skip more than two.

The fourth location in each entry ($nB+3) is the string breakpoint pointer. If it is not zero, then it is assumed that the right half points to (addresses) the start of an ASCIZ string. It had better. The left half doesn't matter. This ASCIZ string is then scanned by the input scanner, just as if you were typing those characters on you keybourd, every time you stop at this breakpoint. This means that if you want to display a certain location each time you hit a certain breakpoint, you can put in the appropriate location a pointer to an ASCIZ string consisting of FOO; (to open location FOO). When the string runs out, Raid takes input from the keyboard. If you have been paying attention, you are probably asking yourself "how do I type control bits?". The answere is, you don't. Fortunately, however, you don't have to. You can, instead of using control bits, use alt-mode. One altmode preceding a character, has the same effect as typing that character with <ctrl 1>. Two alt-modes is the same as <ctrl 2>, and three alt-modes is the same as <ctrl 12>.

MULTI-STEP SPECIAL CONSIDERATIONS:::::::::::

When the multi-stepper gets to a subroutine call or return instruction, is displays a big star and pauses, waiting for input. If you type S it steps, if you type X it executes. Additionally, if you type one of these two things, and control bits are on, special things happen: if you type S with <ctrl 1>, then Raid will no longer stop on subroutine calls, but will always step them. If you type S with <ctrl 2>, then Raid will no longer stop at subroutine returns. If you type S with <ctrl 12>, then both things happen. If you type X with <ctrl 1>, then Raid will no longer stop at subroutine call, but will always execute them. If you type X with <ctrl 2>, Raid will no longer stop at subroutine returns. If you type X with <ctrl 12>, both things happen. This state of affairs remains in effect until you stop the stepping, or change it in the same way you set it, except that it is clear at the start of each new multi-step command.

AD 650 016

ON-LINE DEBUGGING TECHNIQUES: A SURVEY

Thomas G. Evans, et al

Air Force Cambridge Research Laboratories
L.G. Hanscom Field, Bedford, Massachusetts

January 1967

AFCRL-67-0080
JANUARY 1967
INSTRUMENTATION PAPERS, NO. 124

AIR FORCE CAMBRIDGE RESEARCH LABORATORIES

L. G. HANSCOM FIELD, BEDFORD, MASSACHUSETTS

On-Line Debugging Techniques: A Survey

THOMAS G. EVANS
D. LUCILLE DARLEY

OFFICE OF AEROSPACE RESEARCH
United States Air Force

AFCRL-67-0080
JANUARY 1967
INSTRUMENTATION PAPERS, NO. 124

DATA SCIENCES LABORATORY PROJECT 4641
AIR FORCE CAMBRIDGE RESEARCH LABORATORIES
L. G. HANSCOM FIELD, BEDFORD, MASSACHUSETTS

On-Line Debugging Techniques: A Survey

THOMAS G. EVANS
D. LUCILLE DARLEY *

* Bolt, Beranek, and Newman, Inc., Cambridge, Massachusetts

Reprinted from PROCEEDINGS FALL JOINT COMPUTER CONFERENCE,
Vol. 29, pp. 37-49, 7-10 November 1966

OFFICE OF AEROSPACE RESEARCH
United States Air Force

ON-LINE DEBUGGING TECHNIQUES: A SURVEY

Thomas G. Evans

Air Force Cambridge Research Laboratories, Bedford, Massachusetts

and

D. Lucille Darley

Bolt, Beranek, and Newman, Inc., Cambridge, Massachusetts

INTRODUCTION

One consequence of recent interest in the development of large-scale time-sharing systems to provide on-line computer access to a large number of users has been the widespread realization that the usefulness of such a system is critically dependent on the quality of the software provided to facilitate the interaction between user and machine. In particular, one area of critical importance for effective utilization of such a system is that of facilities for program debugging. In view of the important role they play, surprisingly little attention has been paid to the development of facilities to aid in the process of on-line program debugging. Furthermore, much of the work in this field has been described only in unpublished reports or passed on through the oral tradition, rather than in the published literature. The purpose of this paper is to survey the existing work in this area and discuss some possible extensions to it, with the dual goal of acquainting a wider public with currently-existing techniques and of stimulating further developments.

What, precisely, is the intended scope of this paper? First, we are concerned here only with debugging activities taking place in an on-line environment, with the user communicating "conversationally" with his computer by means of, typically, a keyboard (or perhaps a display device and light-pen). Inevitably, there exists overlap between on-line and batch-processing debugging techniques, but our concern here is with the former. Second, we are concerned with *program* debugging; one can, of course, view a wide range of computer use as the debugging of something or other; for example, a numerical method or a physical or economic model. On occasion, of course, this line can be difficult to draw, but we intend to restrict ourselves to activities concerned with the discovery and elimination of program "bugs," in the usual sense, from programs written in typical assembly and higher-level languages and to the "subject-matter-independent" facilities provided to an on-line user to assist in this process.

Why do we place such stress on on-line debugging? Is there really so much difference from debugging in a batch-processing mode? Yes, we think so. One can, of course, ignore the conversational aspect of a time-sharing system and treat it as simply a remote-console job-initiation system. However, in doing this, one is neglecting a potentially very powerful tool—the capability (mediated through suitable debugging aids) for a very selective and close control over the execution of portions of one's program and for the examination of intermediate results, together with the possibility of making on-the-spot changes based on them, as desired. These virtues of on-line access have been praised many times, of course (and debugging is only one activity aided by such access—some on-line uses are so dependent on this type of interaction that they simply have no batch-processing counterparts). We merely wish to add that these benefits for debugging are not automatic results of providing on-line access; as in other aspects of the appearance of on-line systems to their users, careful design of the facilities provided and the conventions for their use pays immense dividends in usability.

Console debugging was common before batch-processing monitors were ever heard of. What's so new about on-line debugging? Nothing, really; current on-line debugging techniques are the result of a gradual development from the days when debugging at the computer console was the norm, as it has remained for small computers over the years. Debugging methods based on single-stepping through parts of a program and on examination and modification of memory registers by means of console lights and switches were the natural precursors of today's more sophisticated techniques, and there is no sharp dividing line at any stage of the progression. Perhaps the critical step was the replacement of console lights and switches by some typewriter-like device as the principal means of communication between user and machine. This permitted the convenient interposition of suitable system programs to facilitate communication between the user and his program. At first they permitted him to examine and modify register contents in typed octal instead of the binary of lights and switches. At a later stage in the development they allowed him to associate symbols with locations in his program and to debug in terms of them, and still later to debug entirely in terms of the original symbols of his assembly-language or higher-level-language programs. The capabilities in this area of current debugging programs will be discussed below. Similarly, a development toward increasing sophistication in the user's control of the flow of his program, as well as in other areas, has taken place and will also be discussed later.

What is the relationship of on-line debugging to time-sharing? On-line debugging (and on-line use of computers in general) is related to time-sharing only in the sense that provision for on-line access to a machine powerful enough for certain classes of problems may be economically feasible only in a time-sharing environment. Furthermore, it is reasonable to expect that many advances in on-line debugging will arise from the communities of users that have already begun to assemble about the currently-existing large-scale time-sharing systems, as well as from the expenditure on system programming that the existence of such communities makes economically justifiable. However, many of the debugging features we shall be discussing had their origins in work with small machines before the advent of time-sharing systems.

In a survey of on-line debugging, a problem of emphasis arises: one might try to convey some of the flavor of the use of typical currently-available techniques to the reader unfamiliar with any existing on-line debugging system; alternatively, one might try to examine and compare in some detail the most important features of the existing systems. We have resolved the problem by attempting both.

The second section of this paper is devoted to a consideration of the principal features of past and present on-line debugging systems known to us, together with some remarks on implementation, on use of displays, and on some implications of the requirements of debugging systems for compiler construction and for hardware. We make no claim of exhaustive coverage. We have discussed those systems which incorporated features which seem to us to have been interesting or significant contributions to the present state of development of on-line debugging.

The third section attempts to impart some "feel" for current on-line debugging methods through two annotated examples. One represents a session devoted to debugging a program written in a typical (but nonexistent) assembly language; the other, a program written in a representative (also nonexistent) algebraic-type language. The examples are idealized in that no one present system contains all of the capabilities illustrated (or uses precisely the set of communication conventions we have adopted), but every feature shown is present in some existing system.

The concluding section contains a few final comments of a general nature.

SURVEY OF EXISTING SYSTEMS

Assembly-Language Debugging

We shall first consider facilities to aid in the debugging of programs written in assembly language.

We have made no extensive effort to disentangle all the threads of the earliest efforts at developing typewriter-based debugging programs. However, the early program which had the greatest influence on subsequent developments was that of Gilmore [1] for the TX-O computer at Lincoln Laboratory in 1957. It was the first in a series of closely-related and successively more elaborate debugging programs, including UT-3 [2] and FLIT [3] for the TX-O (after it was moved to MIT), and DDT [4,5] for the PDP-1 at MIT. FLIT, in particular, was a notable accomplishment, embodying capabilities on which much subsequent work with on-line assembly-language debugging has been based. With FLIT, for the first time, it was possible for the user to examine and modify his program in terms of the symbols used in his source program and, in fact, to examine and change the contents of registers in a form almost identical to that used in the corresponding assembly language. Furthermore, while some earlier typewriter programs had permitted one-instruction-at-a-time tracing of a program, by analogy to the console single-step switch familiar to their creators, FLIT introduced what is perhaps the central notion of interactive debugging, that of a user-controlled breakpoint. This technique, which we shall see illustrated in both assembly-language and algebraic-language debugging in a later section ("Examples—Two Debugging Sessions"), consists of permitting the user to specify (symbolically, typically) a point or points in his program at which he wishes to interrupt its flow and return to the debugging routine, which at entry stores the state of the live registers to permit subsequent continuation from the breakpoint, then permits the user to examine the state of his program at that point and make changes, if he wishes, before continuing. All that is required is that the debugging program save the user's instruction at the desired breakpoint location and plant in its place a suitable transfer to itself. The effectiveness of the technique is dependent, of course, on the ease to the user of placing and removing breakpoints and on the quality of the facilities for examination and modification available to him while at a breakpoint. With judicious use, the breakpoint can be a very flexible tool, giving the user great selectivity in the degree of fineness of his examination of a portion of a program. In the hands of an experienced user, it can permit quite rapid isolation of many types of program error. Here, as in other aspects of on-line work, convenience is critical. The user with only "examine and modify" capabilities available to him could, of course, get the effect of breakpoints by inserting transfer instructions to appropriate inserted code, but the convenience and freedom from elaborate bookkeeping so important to the "iterative" use of breakpoints described above are lost.

FLIT was a program for a one-of-a-kind machine, the TX-O. Consequently, it never became well-known outside its user community at MIT. It was through DDT (written at MIT soon after FLIT as its counterpart on the PDP-1 and embodying much the same set of capabilities, including those sketched above) that these notions were extensively spread about as the PDP-1 became a relatively widely used machine. In this way, FLIT and DDT became the acknowledged source of a large portion of the assembly language debugging programs in the major currently operating time-sharing systems possessing such facilities.

One of the most important characteristics of FLIT and DDT was the care devoted to the design of the typing conventions. Single-letter commands and a structure in which frequently desired states could be reached easily from the present one (e.g., look at the contents of the current register ±1, look at the contents of the register addressed in the current register) minimized typing and aided rapid interaction. Similarly, convenient ways of typing the contents of a given register in alternate formats (e.g., symbolic, decimal, octal) were provided.

Starting with these capabilities, extensions have been made in a number of directions in more recent work. We shall discuss some of these. With the capability for input of machine instructions in symbolic assembly-language form, DDT is already nearly an "on-line assembler," suitable as the sole tool for on-line writing and testing of small programs. With this use in mind, Edwards and Minsky [6] added an "undefined symbol" capability to DDT. In conventional DDT, input of a line of code involving a symbol not already defined by the user results in an error message. In their version, it results in a special symbol table entry. Such entries are linked together, and when the symbol is ultimately defined by the user its previous occurrences are filled in appropriately. This capability has also been included in the assembly-language system [7] of the Berkeley time-sharing system (SDS 940).

DDT permits the user unlimited freedom to patch his program arbitrarily by inserting whatever he likes in some available space, then planting a transfer to this insertion in his program wherever he desires. This very freedom, unfortunately, can lead to situations in which debugging of complex programs ultimately bogs down in a morass of patches on patches. Furthermore, even when a highly patched program has finally been made to perform satisfactorily, the road to a corresponding "cleaned-up" symbolic version of the program can still be a very long and error-susceptible one. We know of two efforts to incorporate at least partial solutions to these bookkeeping problems into assembly-language debugging systems. In one approach, followed by Lampson [8] in the design of one version of the assembly-language debugging facilities in the Berkeley system, the user requests the insertion of a specified piece of symbolic code starting at a specified symbolic location in his program (or deletion of a portion of the existing program, or both). In response to this request, the debugging program performs two distinct activities:

1. It edits the user's changes into his symbolic program stored on the drum.
2. It assembles the user's addition into a "patch area" of core and automatically links the resulting code to the user's program in a straightforward way by copying instructions and inserting transfers, as necessary.

Thus, at each stage of the debugging process, the user's patched binary program in core is "computationally equivalent" to the edited version of his symbolic on drum. At the completion of the debugging session, the user's updated symbolic is stored again among his files.

An earlier approach [9] to the same problem, taken by the present authors in work with an assembly-language debugging system for the M-460 computer at Air Force Cambridge Research Laboratories, is quite different in implementation. Once again, the on-line user presents insertions, deletions, or a mixture of both (again in symbolic assembly language) to the debugging program, using a quite similar set of conventions. Once again two actions are taken:

1. The symbolic changes are stored in a form suitable for entry, along with the original symbolic program, to an editing program at the end of the debugging session. This is automatically done and the user provided with an updated symbolic file. This difference—saving the corrections to do all the editing at one time vs editing for each correction, as in the system discussed previously—is a thoroughly trivial and inessential one.

2. Instead of a patch being made corresponding to the user's change, the part of the program affected by the change is relocated appropriately in core. If the change is an insertion, for example, the new code is assembled into the space left vacant by the relocation of the program from that point on. This relocation process is possible only because the relocation information resulting from the assembly of the user's program, in addition to being used by the relocating loader, is collected by it into a list structure which is used by the debugging program each time a program change is called for by the user, then updated accordingly. The symbol table passed by the assembler to the debugging program must also be updated each time. Thus the idea of "patching" disappears completely. This relocation process can be rather time-consuming on large programs, but has certain compensating advantages over the (quite fast) "automatic patching" approach of Ref. 8. In particular, it avoids the two drawbacks of his system listed by Lampson:

a. In situations in which location of words in core relative to each other is important (for example, subroutine calls picking up arguments from following locations), the patched binary and the edited symbolic may behave differently.

b. The automatic patching process leaves core in a rather confusing state, which may require relatively frequent reassembly for readability. For example, the user who wishes to insert a breakpoint at an instruction inserted during one of his previous modifications must track down the present location of that instruction by finding and following out the patching. Thus, much of the advantage of automatic patch insertion could well be nullified.

Any evaluation of the two approaches must balance the added program complexity and computation time required by the relocation approach against the possible cost in inconvenience to the user of the above difficulties (or, alternatively, the cost in computation time of the additional assemblies that may be required to preserve sufficient readability).

One further extension to DDT in more recent work pertains to the use of breakpoints. In addition to the flexibility in the placement and moving of breakpoints which is already present in DDT, a facility has been added in a number of debugging programs (including those for the SDC time-sharing system,[10] the DEC PDP-6,[11] and the M-460 at AFCRL) permitting the user to make the breakpoints conditional; when the breakpoint location is reached, some test previously supplied on-line by the user is executed to determine whether the break is to be made (that is, control turned over to the user) or whether execution of the user's program should continue. This technique permits still greater selectivity; the user can run his program till some specified condition prevails at a specified point, then examine the program state in whatever detail he wishes. The SDC system gives the user a choice of a number of built-in conditions; the other two permit the user to insert an arbitrary piece of assembly-language code as the break test associated with each breakpoint. Ideally one would like to combine "canned," easily specifiable tests for certain common situations with the capability of writing arbitrary tests when desired. DDT, incidentally, had a rudimentary but often useful form of the conditional breakpoint which has been preserved in several later systems; upon insertion of a breakpoint, the user may specify (simply by preceding the command with a number *n*) that the break is not to occur until the *n*th time that that point is reached in the execution of the program.

A possibility we have not yet examined, but which forms a basic tool of some early on-line debugging programs, is that of instruction-by-instruction tracing. More sophisticated versions of such tracing, with considerable flexibility available to the user, have been incorporated in debugging packages for batch-processing use, but such tracing features have typically been omitted from more recent on-line systems in favor of the breakpoint, on the grounds that tracing represents a failure to make the most of the capability for intensive interaction possible in such a system and, at best, tends to produce considerable irrelevant printout, a serious consideration for an on-line user. However, it seems to us reasonable to provide some tracing capabilities in an on-line system, especially since they can share much of the machinery already provided for breakpoints. The user should be able to specify a location in his program and ask either for the printing of certain information, for control, or for a combination of both whenever that program point is reached (and a specified condition is satisfied). Currently no assembly-language debugging system appears to have quite this full capability, though PDP-6 DDT [11] and the SDC DBUG package [10] are close. Both are limited in the amount of information that can be specified in advance to be printed at a break—in the PDP-6 DDT to one register and in DBUG to one register or a live register dump or a dump of some block of registers. Furthermore, as mentioned above, DBUG does not permit the composition of elaborate conditions for a break to occur.

Another desirable feature not widely found in current assembly-language debugging systems is extensibility, in the sense of the capability for conveniently defining complex debugging operations in terms of the available primitives. The most general existing facility of this type appears to be that described in Ref. 8, where the macro-expansion capability of the assembler used to process input to the DDT lends itself quite naturally to this purpose.

Programs of the DDT family have many useful features in addition to the ones we have described. As one example, it is typically possible to conduct a search between specified limits in core for all words matching a given word in the bits specified by a given mask.

Higher-Level-Language Debugging

When we turn to the examination of on-line debugging facilities for programs written in higher-level languages comparable to those we have considered for assembly-language programs, we find that, broadly speaking, a close analog of almost every principal assembly-language debugging technique exists in at least one debugging system pertaining to some higher-level language. However, on-line debugging facilities for higher-level languages are in general less well-developed and less widely used (relative to the use of the languages) than their assembly-language counterparts. In part, this situation is probably a consequence of the wide diversity of languages in this class; probably it is still more a result of the fact that the small machines on which the assembly-language techniques originated and were cultivated were typically considered too small to support higher-level language compiling systems and were programmed almost exclusively in assembly language. Thus work in on-line debugging of higher-level languages is of comparatively recent origin. We shall be examining debugging systems for relatively few languages in relatively few on-line computing systems. This is not to say that much more on-line debugging (in the sense that the user at a remote console starts his program, examines the final results or diagnostics in essentially the manner of batch processing, edits his program, and tries again) is not taking place in these and other systems with these and other higher-level languages. However, we are concerned especially with efforts to obtain systems which permit the on-line user something like the flexible control over the execution of his program and the capability of examining and modifying it that are available to the one-line user of the assembly-language debugging aids we have discussed.

The language for which perhaps the most effort has been expended in the development of on-line debugging aids is the list-processing language LISP 1.5. However, no discussion of these debugging features has appeared in the literature; they are far from completely described even in internal memoranda. The first two full-scale on-line implementations of LISP were those for the MAC system [12,13] (a modification of the batch-processing LISP system for the IBM 7094 to run under the MAC time-sharing system) and for the M-460 computer at AFCRL. Subsequently, on-line LISP 1.5 systems have been created for the SDC time-sharing system,[14] the Berkeley system,[15] the DEC PDP-6,[16] and the DEC PDP-1 at Bolt, Beranek, and Newman, Inc.[17] We shall discuss only the debugging features of the MAC and M-460 systems, as the later systems contain essentially no debugging aids not already present in these.

First, the extensive tracing facilities of the LISP system were made accessible to the on-line user. Later, they were extended and made conditional in both systems. An editing program—not a conventional text editor but a program permitting the user to modify the list structure in which LISP functions are stored for interpretation—was introduced by Martin into the MAC LISP system and soon modified for use in M-460 LISP. This editor proved to be a powerful tool, permitting quite easy program modification in many cases. Conditional breakpoints (insertible at any point in a LISP function definition) were introduced into the M-460 LISP system by one of the present authors—apparently, along with the introduction of breakpoints into the SDC IPL-V system by Weissman,[18] their first use in higher-level language debugging—and soon after incorporated in MAC LISP. Conditional breakpointing and tracing have proved quite powerful for LISP debugging, as it is possible to use the full capability of the LISP language for the on-line composition of the conditions. Thus one can easily express an elaborate logical condition for which the counterpart in assembly language might be quite complex. Furthermore, by "canning" a few useful special predicates for use in writing conditions, even more selectivity in suppressing irrelevant tracing and breakpoints can be attained. For example, in M-460 LISP there is a machine-language LISP function which examines the interpreter's pushdown list to answer the question: "At this point in the execution of the program are we inside a call to function X?" Incidentally, the ability of LISP to handle recursion has proved very useful in debugging—the full capability of the LISP system is available at a breakpoint inside a function being executed. With some care, it has been possible, for example, to find a bug while at a breakpoint in running a test case, call the editor to make a correction, run the program on a simpler test case to verify the correctness of the change, then resume execution of the original test case from the breakpoint (without the addition of any special machinery to the system for saving and restoring a program state, etc.).

At this point, it should be mentioned that both LISP systems mentioned contain both an interpreter (of LISP functions stored as list structures) and a compiler (of LISP functions into machine code), and that interpreted and compiled functions may be quite freely intermixed. The existence of the interpreter made the implementation of the debugging facilities described above relatively simple. For example, insertion of breakpoints at arbitrary locations in a LISP program is readily implemented by modifying the list structure corresponding to the program so as to call the breakpoint-handling routine appropriately. In addition, interpretation has the advantage that various types of user errors may be conveniently detected at run time. A further advantage in this case is that LISP is precisely a language for manipulation of list structures so the breakpoint insertion routine, among others, could itself be written entirely in LISP. On the other hand, the feature in LISP that permits tracing of entries (with printing of arguments) and exits (with printing of values) of specified routines applies to both compiled and interpreted routines. However, the usual mode of operation in the systems mentioned has been to debug interpretively, then compile the debugged programs in cases where the great speed advantage to be gained by compiling is important. In general, interpretation presents similar advantages for other higher-level languages, and we shall see below that it has furnished the basis for on-line debugging systems for other languages, as well. We shall also mention two systems which work with a compiled program. Then we shall consider one effort to design a system combining interpretation and compilation, with the intention of combining the speed advantage of compiled programs with the ease of modification that comes with interpretation.

The well-known QUIKTRAN system [19] is based on interpretation of FORTRAN statements. The FORTRAN program under debugging may be modified freely by insertion and deletion of statements. A form of nonconditional breakpoint capability is included in the sense that a statement can be inserted at any point in the program which, when reached, has the effect of transferring control to the user. Capability for examining and modifying variables is present, as well as a variety of modes of tracing (print all assignments to variables in a given portion of the program, all assignments to selected variables, all control transfers within a specified region, etc.). Furthermore, extensive run-time diagnostics made possible by the interpretive mode are provided, and several unusual "bookkeeping" features, similarly based on interpretation, are available, such as the AUDIT command, which generates information as to which portions of the program were never executed, which variables were never set, or set but never used, during a given execution of the program.

Another on-line debugging system based on interpretation is that for IPL-V in the SDC time-sharing system.[18] It contains (nonconditional) breakpoint and tracing capabilities similar to those sketched above for LISP.

The FORTRAN debugging system [20] for the Berkeley time-sharing system and the MADBUG system [21] for the debugging of MAD language programs are very similar in their debugging capabilities, though different in overall scope: MADBUG contains a set of editing facilities as well, while editing of FORTRAN symbolic programs is carried out in the Berkeley system by use of a general-purpose editing routine present in the time-sharing system. In both cases debugging is performed on a compiled version of the program, and the user can readily ask for the values of variables and change them. Breakpoints (nonconditional, as it happens) at any specified statement (in the Berkeley FORTRAN, labeled statement) may be inserted and deleted. However, no facility is provided to modify portions of the user's program (in both systems, a user familiar with the code produced by the compiler could, of course, use the available assembly-language debugging facilities to make such local modifications). The only way to make program changes is to edit the symbolic version and recompile the whole program.

The notion of an "incremental" compiler, in which only those portions of a program to be changed need to be recompiled, has been frequently discussed; Lock [22] at California Institute of Technology has given a detailed sketch of the design of a system with such capabilities. The notion is to compile each program statement separately and place the resulting code, together with a copy of the symbolic form of the statement and certain pointers and other information, depending on the type of statement, in a contiguous block of core. These blocks would be linked together in lists. Since the language in question is ALGOL, in which "statement" is a recursively defined concept, one has a list structure (lists with elements which are lists, etc.) instead of the one-level list of statements one would have with, for example, FORTRAN. Insertion and deletion at the statement level proceed straightforwardly by modification of this list structure. Control is returned to a monitor between statements (this is a property of the code generated for each statement) permitting, among other things, breakpoint capability at the statement level (though the author proposes simply a single-statement mode of operation modeled, apparently, on single-step-switch machine-language debugging). The scheme is interesting and quite ambitious. It remains to be seen whether the organization based on compiled statements with interpreted flow of control between them leads to significantly faster execution times than pure interpretation with a well-designed internal representation such as that of QUIKTRAN. One possible modification in the scheme would be to arrange things so that the code in the block corresponding to a statement transfers, not back to the executive, but to what at that point is the correct next statement. The executive would maintain a table of these transfer locations and "breakpoint" them, so to speak, whenever this was called for (as a result, for example, of a breakpoint request by the user). Thus, at the cost of some additional complexity in the executive, almost all the speed advantage of full compilation would be realized with no loss in the capabilities available to the user.

It seems appropriate to mention at this point a class of languages of which JOSS, BASIC, and TINT are the best-known examples, even though a principal characteristic of these languages, especially the first two, is their lack of anything which looks like the tracing or breakpoint features we have discussed. These are "small" languages designed precisely for easy learning and convenient on-line use for problems requiring a numerical computing capacity somewhere between a desk calculator and the typical "FORTRAN + large computer" installation. In all three languages, insertion, deletion, and modification of statements is extremely easy; since this is so, the effect of tracing and breakpoints can be achieved for the small, relatively simple programs in question by insertion of print and halt statements respectively. Thus, at the borderline of the class of languages and associated debugging tools that we have discussed earlier we find a class of languages that have rather effectively transcended the need for such tools by careful design and ruthless simplification of language structure corresponding to the setting of limited objectives for the range of usefulness of the language.

Hardware Aspects

There are many points of interaction between computer hardware design and the design of software debugging facilities. We shall mention just two:

1. The capabilities of user consoles have a great impact on the range of debugging facilities. Suppose the user is provided with a display device in addition to (or instead of) his keyboard. One may use this added capability relatively conservatively as an extension of facilities already present. For example, the Edwards-Minsky version of DDT already mentioned [6] used a display to permit the user much more rapid and convenient examination (in symbolic and octal) of his program in core than would have been feasible with a typewriter alone. Programs to display core in octal already existed more than 10 years ago.[23] Other, more radical uses of display devices in debugging are now being investigated. Flow-chart languages, where programs are created on-line by generating a flow chart with a light-pen, are being studied at Lincoln Laboratory [24] and at RAND.[25] A dynamic display of the program state at any point in terms of the flow chart is expected to be a useful debugging tool. Other work at Lincoln Laboratory [23] is directed toward dynamically mapping out on a display device the behavior of a more conventionally-constructed program by means of a flow diagram, which is again expected to be a useful debugging aid.

2. Another area of contact between hardware and debugging is involved with trapping. Program-controllable facilities for trapping on certain machine conditions give promise of being a very important debugging aid. The TX-2 computer at Lincoln Laboratory, for example, has recently been provided with a quite powerful interrupt system of this nature, which has been made accessible to the on-line user through commands to a DDT-like program.[26] The user may ask for a trap on any combination of a number of conditions, such as a store into a specified register, execution of an instruction at a specified location, or execution of any skip or jump instruction. The debugging program handles the interrupt and reports the relevant information to the user.

EXAMPLES: TWO DEBUGGING SESSIONS

Assembly Language Debugging

Assume we wish to debug the following program (written in a typical—but mythical—assembly language), which is meant to perform a simpleminded exchange sort on a table of five numbers.

```
sort    pze
        call    .readf
        bci     data1
        pze     table
        lix     nm2,1
loop    lda     table+1,1
        sub     table,1
        sma
        jmp     ok
        lda     table,1
        ldq     table+1,1
        sta     table+1,1
        stq     table,1
ok      tiz     .+2,1
        jmp     loop
        ret     sort
nm2     pze
table   bss     5
        end
```

(This is admittedly a trivial program which should not need the elaborate debugging facilities we have discussed; however, it should serve to illustrate the application of these techniques in an otherwise reasonably realistic context.)

We assume that, previous to this debugging session, we have stored our symbolic program as a file, either by reading in cards or paper tape, or by typing the program in directly from our console. We then assembled our program from the file and created a new file containing the loadable form of the program as well as the symbol table. We omit describing these procedures in detail, for the process of controlling an assembly on-line and the error diagnostics received are much the same as assembling off-line (with, however, the advantage that any errors detected by the assembler can be corrected immediately). We also assume that a test case file called data1 (which will be read by our program) has been written. For definiteness, assume it consists of the numbers 3, 5, 2, 1, 4 in that order.

The loading system has brought our program into core and has left us in contact with the debugging system, which it has supplied with the symbol table for our program. We immediately attempt to execute the program by typing:

```
G sort
```

(This calls sort, which is written as a subroutine.) The debug system responds with a carriage return, indicating completion of our program. We type:

```
P table; table+4
```

which prints:

```
table   3
        5
        2
        1
        4
```

That is, the table we input is unchanged. Examining our program, we note that the instruction at ok performs a test for the end of a pass through the table. It seems a plausible instruction to monitor, so we insert a breakpoint (number 1) there:

```
B1 ok
```

and then execute the program again. The computer responds with:

```
ok
```

indicating that it has reached the breakpoint. At this point we can examine whatever registers we wish, including live registers, by typing the symbolic name plus a tab. The computer will respond with the contents of the register in symbolic format, tab, and wait for us to modify the contents of the register. If we don't wish to, a carriage return signifies this. Index register 1 is important in our program, so we examine it:

```
I1      0
```

We see that this value is incorrect. The instruction at loop-1 supposedly loads index register 1. We check it:

```
loop-1    lix nm2,1
```

This is apparently correct, so we check nm2:

```
nm2     0
```

This is our error. We neglected to initialize nm2. We give the following command:

```
C nm2
nm2       oct 3
```

which says to change the contents of the register labeled nm2 to whatever follows; in this case, the register is to carry the same label but contain the number 3, the length of our table minus 2. Our program is physically changed in core, and the necessary information concerning this change is saved so it can be given to an editing program at the conclusion of our debugging session. We remove the breakpoint inserted earlier by typing:

```
B1
```

and then start the program again. Again the debug system carriage returns. We now check the contents of the table, as before:

```
table   1
        3
        5
        2
        4
```

Obviously not all of the ordering is correct. Perhaps it would be useful to reinsert the breakpoint at ok, since it is the instruction immediately following the instructions that switch the contents of registers. However, we would like to break here only if an exchange did occur, and at the break we would like to print the contents of the two registers in the table that were switched. This allows us to monitor the successive changes in the table, so we can see at what point something goes wrong. We insert this type of breakpoint by:

```
B1 ok: P table,1; table+1,1: C
B1C
lda        table+1,1
sub        table,1
spa
end
```

The first line indicates that breakpoint 1 should be inserted at ok and when that point is reached two things should be printed out: the contents of the register at (table + the contents of index register 1), and the register at (table+1 + the contents of index register 1). C means to continue after the printing without transferring control to the user. The second line indicates that we are going to give a condition for breakpoint 1, and the next three lines are the condition, with the instruction following the spa (skip on positive AC) the break branch and the instruction following that the proceed branch. With each breakpoint the debug program associates three registers that are used in determining if a break should occur when the breakpoint has been reached. Initially, and each time a breakpoint is removed, the three registers appear as:

```
nop (no operation)
(return for break to occur)
(return for no break to occur)
```

With this arrangement, a break occurs each time the breakpoint is reached. When a condition (other than a single skip instruction) for a break is entered, as we did above, the nop instruction is automatically changed to a jump to a patch region where the code we supply is inserted. The first register following this code is assumed to be the break condition and a jump is inserted there to the first return. The second instruction following the code is set up as a jump to the second return.

We now execute our program again and get the following results:

```
ok
table+2     1
table+3     2

ok
table+1     1
table+2     5
ok
table       1
table+1     3
```

which is correct as far as it goes, but after the above printout the debug system carriage returns, again indicating that our program has returned. Looking at our program again, we see that we left out the outer loop in our coding and are making only one pass through the table. We make the following changes:

```
I sort+3
loop2       stz switch
I loop+3
            idx switch
I ok+1
            lda switch
            sza
            jmp loop2
I nm2
switch      pze
```

The first change inserts an instruction labeled loop2 after sort+3 to initialize the register labeled switch, which is at this point undefined. The second change inserts an instruction after loop+3 to increment the contents of switch. The third change is to insert three instructions after ok+1. They again refer to switch and also to loop2, which was defined in the first change. The fourth change defines switch, and at this point all references to it are automatically filled in.

We now run our program again after removing the breakpoint and type out the results as above. This time the table is fully sorted. At this point in our debugging session we decide that we now have a working program in core. To get a clean symbolic version, we give a command to the debug system to supply the changes which we have made (and it has kept) to an editing program, along with our original file. The editing process takes place and the new file is written and given whatever name we have specified. Our final (purportedly debugged) program looks like this:

```
sort        pze
            call     .readf
            bci      data1
            pze      table
loop2       stz      switch
            lix      nm2,1
loop        lda      table+1,1
            sub      table,1
```

```
        sma
        jmp     ok
        idx     switch
        lda     table,1
        ldq     table+1,1
        sta     table+1,1
        stq     table,1
ok      tiz     +2,1
        jmp     loop
        lda     switch
        sza
        jmp     loop2
        ret     sort
nm2     oct     3
switch  pze
table   bss     5
        end
```

Higher-Level-Language Debugging

Our example of on-line debugging of a higher-level-language program will be shorter than the preceding assembly-language example, since we simply wish to show that the facilities exhibited there for control of program flow and for examination and modification of program and data have their counterparts at other levels of language as well. Correspondingly, our program example is even more trivial; it is the same exchange sort programmed in a typical (but again mythical) algebraic language with the same sort of (admittedly implausible for a program of this simplicity) errors.

Again we assume our program has previously been made into a symbolic file, then compiled and loaded. We shall test it on the same file (data1) as we used in the previous example. Our program reads as follows:

```
program  sort;
         array table (5);
         readfile(data1,table);
   loop: for i←1 step 1 unit 4 do through last;
         if table(i) ≤table(i+1) go to last;
         begin table (i)←table(i+1);
               table(i+1)←table(i)
         end;
   last: continue;
         go to loop;
finish
```

(readfile is a system routine that we call to fill the array named table from the file named data1). We run our program (named sort) by typing G sort, as before. In this case, we conclude after a respectable interval that our program is looping. By striking the interrupt key, we return to the debugging program. We realize that we have failed to provide a test to escape from the program after a pass through the table generates no exchanges. We therefore make several additions to our program, as follows:

```
I loop −1
;init: switch←false;
```

(which means insert the labeled statement setting the Boolean variable switch to false after statement loop −1, that is, then "readfile" statement).

```
C last+1
;if switch then go to init else exit;
```

(which means insert the statement testing the variable switch instead of the statement last+1, that is, "go to loop"). And finally:

```
I loop+2,2
;switch←true;
```

(which means insert the statement setting the variable switch to true after the second statement of the compound statement at location loop+2).

To verify the last change, for example, we can type

```
E loop+2
```

which prints:

```
begin table (i)←table(i+1);
      table(i+1)←table(i);
      switch←true
end
```

Now we try running our program again. This time it terminates and returns control to the debugging program. However, when we examine the results by typing:

```
P table(1);table(5)
```

we get:

```
table(1)=1
table(2)=1
```

At this point we interrupt the printout, since our answers are clearly in error. We insert a breakpoint at the statement labeled last and add the condition that we break only if switch has been set to true, by typing:

```
B1 last
B1C switch
```

We run the program again and get the breakpoint printout:

```
last
```

We examine the indexing variable:

```
i        2
```

So we look at:

```
table(2)      1
table(3)      1
```

This tells us we're doing the exchange wrong. We see that we are destroying table(i) too soon, and correct this by typing:

```
I loop+2,0
;tem←table(i);
```

and

```
C loop+2,3
;table(i+1)←tem;
```

and rerun our program. This time, when we examine table, it is properly ordered. We terminate the session, as before, by passing the accumulated corrections to the editing program, which updates our symbolic. The final version of our program looks like:

```
program sort;
        array table(5);
        readfile(data1,table);
  init: switch←false;
  loop: for i←1 step 1 until 4 do through last;
        if table(i) ≤table(i+1) go to last;
        begin tem←table(i);
              table(i)←table(i+1);
              table(i+1)←tem;
              switch←true
        end;
  last: continue;
        if switch then go to init else exit;
finish
```

Once again, in conclusion, we stress that the programs used in the examples of this section were not intended as representative of those for which such on-line debugging facilities are necessary or even appropriate, but rather as uncluttered vehicles for some simple illustrations of the use of these facilities.

SOME FINAL REMARKS

Very little data seems to exist on the relative efficiency of on-line program debugging versus debugging in a batch-processing mode, though Ref. 27 represents a first effort in this direction and will presumably be followed by others. Meanwhile, we can only record our subjective impressions of a quite widespread enthusiasm for the utility of on-line debugging facilities among those with whom we have discussed the subject.

What are some criteria for a good interactive debugging system (for an experienced user)? We shall try to abstract some (perhaps platitudinous) principles from the wide variety of systems considered above:

1. The user must have flexible control over the execution of his program. He must be able to specify this control in terms of the natural units, small and large, of the language in question and be able to carry this control down to the finest level of detail, if required (a single instruction in assembly language, or single noncompound statement in an ALGOL-type language).
2. The user must be able to examine and "incrementally" modify both data and program at any time and do so in terms of the notation of the language of the program.
3. The conventions of the debugging control language should be designed to minimize typing and should convey information to the user as concisely as is compatible with rapid comprehension.
4. Automatic updating of a user's symbolic file "in parallel" with modification of the in-core representation of his program should be possible, to eliminate a distinct separate phase of cleanup of the symbolic and redebugging.

Each of these capabilities is now present, as we have seen, to some degree in some current systems. With feasibility thus demonstrated, in the near future we shall presumably see the integration of features taken from these systems into comprehensive on-line debugging systems possessing all the desirable characteristics listed above. Incidentally, this seems to present an opportunity for some valuable voluntary standardization; if the appearance to the user of the debugging system for a given language could be made the same over a number of future time-sharing systems (at least to the degree that the language in question is itself standardized), considerable savings could well be realized. At any rate, this would seem to be an opportune time to consider the possibility.

In addition to consolidation of known techniques into comprehensive, widely available systems, one can also expect the development of a variety of new approaches; in particular, we have mentioned research which seeks to exploit the full capabilities of displays for debugging, as well as the potential value for debugging of flexible programmable interrupt capabilities in computer hardware.

Considerable interest has been shown in recent years in the development of methods for proving that a given computer program has certain properties. If this avenue of research proves successful, we may one day see the virtual elimination or at least diminution in importance of the program debugging process. Until then, debugging will remain a critical phase and potential bottleneck in the effective utilization of computers. It has been suggested [28] that, from a period in which the limitations on computer use were in core size and in sheer lack of enough processor cycles to go around, followed by one of lack of adequate languages, we are now entering an era in which computer use is "debugging-limited." If this is so, the development of improved on-line debugging facilities would seem to be a particularly fruitful and valuable endeavor, as well as a quite fascinating one.

REFERENCES

1. J. T. Gilmore, "TX-O Direct Input Utility System," Memo 6M-5097, Lincoln Laboratory, MIT (Apr. 1957).

2. C. Woodward, "UT-3: A Direct Input Routine for TX-O," Memo M-5001-1, Dept. of Elect. Eng'g., MIT (July 1958).

3. T. G. Stockham and J. B. Dennis, "FLIT—Flexowriter Interrogation Tape: A Symbolic Utility Program for TX-O," Memo 5001-23, Dept. of Elect. Eng'g., MIT (July 1960).

4. R. Saunders and R. Wagner, "On-Line Debugging Systems," *Proc. IFIP Congress, 1956, Vol. 2*, Spartan Books, Washington, D.C.

5. A. Kotok, "DEC Debugging Tape," Memo MIT-1 (rev.), MIT (Dec. 1961).

6. D. J. Edwards and M. L. Minsky, "Recent Improvements in DDT," AI Memo #60, MIT (Nov. 1963).

7. L. P. Deutsch and B. W. Lampson, "DDT Time Sharing Debugging System Reference Manual," Document #30.40.10 (rev.), Univ. of Calif., Berkeley (May 1965).

8. B. W. Lampson, "Interactive Machine Language Programming," *Proc. FJCC,* 1965.

9. T. G. Evans and D. L. Darley, "DEBUG—An Extension to Current Online Debugging Techniques," *Comm. of the ACM,* vol. 8, no. 5 (May 1965).

10. R. R. Linde, "Q-32 Time-Sharing System User's Guide Executive Service: Debugging (DBUG)," TM-2708/390/00, Syst. Devel. Corp. (Apr. 1966).

11. "PDP-6 DDT Manual," Digital Equipment Corp., 1965.

12. W. Martin and T. Hart, "Time-Sharing LISP," Memo MAC-M-153 (rev. 1964).

13. W. Teitelman, "EDIT and BREAK Functions for LISP," Memo MAC-M-264, MIT (1965).

14. S. L. Kameny, "LISP 1.5 Reference Manual for Q-32," TM-2337/101/00, Syst. Devel. Corp. (Aug. 1965).

15. L. P. Deutsch and B. W. Lampson, "Reference Manual—930 LISP," Document #30.50.40 (rev.), Univ. of Calif., Berkeley (Nov. 1965).

16. P. Samson, "PDP-6 LISP," Memo MAC-M-313, MIT (June 1966).

17. D. G. Bobrow et al, "The BBN-LISP System," AFCRL-66-180, Bolt, Beranek, and Newman, Inc., Cambridge, Mass. (Feb. 1966).

18. J. E. Schwartz, E. G. Coffman, and C. Weissman, "A General-Purpose Time-Sharing System," *Proc. SJCC,* 1964.

19. T. M. Dunn and J. H. Morrissey, "Remote Computing—An Experimental System," ibid.

20. C. S. Carr, "FORTRAN II Reference Manual," Document #30.50.50, Univ. of Calif., Berkeley (Feb. 1966).

21. R. S. Fabry, "MADBUG—A MAD Debugging System," in *The Compatible Time-Sharing System, A Programmer's Guide,* 2d ed., MIT Press, Cambridge, Mass., 1965.

22. K. Lock, "Structuring Programs for Multiprogram Time-Sharing On-Line Applications," *Proc. FJCC,* 1965.

23. T. G. Stockham, "Some Methods of Graphical Debugging," to appear in Proc. IBM Scientific Computing Symposium on Man-Machine Communication (held May 1965).

24. W. R. Sutherland, "On-Line Graphical Specification of Procedures," presented at SJCC, Boston, Mass., 1966 (unpublished).

25. T. O. Ellis and W. L. Sibley, "The Grail Project," ibid. (unpublished).

26. T. G. Stockham (personal communication).

27. E. E. Grant, "An Empirical Comparison of On-Line and Off-Line Debugging," SP-2441, Syst. Devel. Corp. (May 1966).

28. M. Halpern, "Computer Programming: The Debugging Epoch Opens," *Computers and Automation*, Nov. 1965.

Unclassified
Security Classification

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)*
Hq AFCRL, OAR (CRB)
United States Air Force
Bedford, Massachusetts 01730

2*a*. REPORT SECURITY CLASSIFICATION
Unclassified

2*b*. GROUP

3. REPORT TITLE
On-Line Debugging Techniques: A Survey

4. DESCRIPTIVE NOTES *(Type of report and inclusive dates)*
Journal Article. Interim.

5. AUTHOR(S) *(Last name, first name, initial)*
EVANS, Thomas G. and DARLEY, D. Lucille

6. REPORT DATE
January 1967

7*a*. TOTAL NO. OF PAGES
20

7*b*. NO. OF REFS
28

8*a*. CONTRACT OR GRANT NO.

b. PROJECT AND TASK NO. 4641-02

c. DOD ELEMENT 62405454

d. DOD SUBELEMENT 674641

9*a*. ORIGINATOR'S REPORT NUMBER(S)
AFCRL-67-0080
IP No. 124

9*b*. OTHER REPORT NO(S) *(Any other numbers that may be assigned this report)*
AFCRL-67-0080

10. AVAILABILITY/LIMITATION NOTICES
Distribution of this document is unlimited.

11. SUPPLEMENTARY NOTES Reprinted from
Proceedings Fall Joint Computer
Conference, Vol. 29, pp. 37-49,
7-10 November 1966

12. SPONSORING MILITARY ACTIVITY
Hq AFCRL, OAR (CRB)
United States Air Force
Bedford, Massachusetts 01730

13. ABSTRACT

In view of the recent widespread interest in computer systems capable of providing on-line access to a large number of users, the development of system software to facilitate such interaction has become increasingly important. This paper makes what is apparently the first attempt to survey past and present work in one such area of particular importance for effective computer utilization, the development of facilities to aid the on-line user in debugging his programs. Various aspects of the appearance to the user of such debugging systems, as well as of their implementation, are discussed. Both assembly-language and higher-level-language debugging techniques are examined, and annotated examples of debugging sessions are included to impart a feeling for the capabilities of current systems.

Unclassified
Security Classification

Unclassified
Security Classification

14. KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
Program Debugging						
Man-machine interaction						
On-line systems						

INSTRUCTIONS

1. ORIGINATING ACTIVITY: Enter the name and address of the contractor, subcontractor, grantee, Department of Defense activity or other organization *(corporate author)* issuing the report.

2*a*. REPORT SECURITY CLASSIFICATION: Enter the overall security classification of the report. Indicate whether "Restricted Data" is included. Marking is to be in accordance with appropriate security regulations.

2*b*. GROUP: Automatic downgrading is specified in DoD Directive 5200.10 and Armed Forces Industrial Manual. Enter the group number. Also, when applicable, show that optional markings have been used for Group 3 and Group 4 as authorized.

3. REPORT TITLE: Enter the complete report title in all capital letters. Titles in all cases should be unclassified. If a meaningful title cannot be selected without classification, show title classification in all capitals in parenthesis immediately following the title.

4. DESCRIPTIVE NOTES: If appropriate, enter the type of report, e.g., interim, progress, summary, annual, or final. Give the inclusive dates when a specific reporting period is covered.

5. AUTHOR(S): Enter the name(s) of author(s) as shown on or in the report. Enter last name, first name, middle initial. If military, show rank and branch of service. The name of the principal author is an absolute minimum requirement.

6. REPORT DATE: Enter the date of the report as day, month, year, or month, year. If more than one date appears on the report, use date of publication.

7*a*. TOTAL NUMBER OF PAGES: The total page count should follow normal pagination procedures, i.e., enter the number of pages containing information.

7*b*. NUMBER OF REFERENCES: Enter the total number of references cited in the report.

8*a*. CONTRACT OR GRANT NUMBER: If appropriate, enter the applicable number of the contract or grant under which the report was written.

8*b*, 8*c*, & 8*d*. PROJECT NUMBER: Enter the appropriate military department identification, such as project number, subproject number, system numbers, task number, etc.

9*a*. ORIGINATOR'S REPORT NUMBER(S): Enter the official report number by which the document will be identified and controlled by the originating activity. This number must be unique to this report.

9*b*. OTHER REPORT NUMBER(S): If the report has been assigned any other report numbers *(either by the originator or by the sponsor)*, also enter this number(s).

10. AVAILABILITY/LIMITATION NOTICES: Enter any limitations on further dissemination of the report, other than those imposed by security classification, using standard statements such as:

(1) "Qualified requesters may obtain copies of this report from DDC."

(2) "Foreign announcement and dissemination of this report by DDC is not authorized."

(3) "U. S. Government agencies may obtain copies of this report directly from DDC. Other qualified DDC users shall request through

____."

(4) "U. S. military agencies may obtain copies of this report directly from DDC. Other qualified users shall request through

____."

(5) "All distribution of this report is controlled. Qualified DDC users shall request through

____."

If the report has been furnished to the Office of Technical Services, Department of Commerce, for sale to the public, indicate this fact and enter the price, if known.

11. SUPPLEMENTARY NOTES: Use for additional explanatory notes.

12. SPONSORING MILITARY ACTIVITY: Enter the name of the departmental project office or laboratory sponsoring *(paying for)* the research and development. Include address.

13. ABSTRACT: Enter an abstract giving a brief and factual summary of the document indicative of the report, even though it may also appear elsewhere in the body of the technical report. If additional space is required, a continuation sheet shall be attached.

It is highly desirable that the abstract of classified reports be unclassified. Each paragraph of the abstract shall end with an indication of the military security classification of the information in the paragraph, represented as *(TS), (S), (C), or (U)*.

There is no limitation on the length of the abstract. However, the suggested length is from 150 to 225 words.

14. KEY WORDS: Key words are technically meaningful terms or short phrases that characterize a report and may be used as index entries for cataloging the report. Key words must be selected so that no security classification is required. Identifiers, such as equipment model designation, trade name, military project code name, geographic location, may be used as key words but will be followed by an indication of technical context. The assignment of links, rules, and weights is optional.

Unclassified
Security Classification

Published: 6/7/66

Identification

Interactive debugging aids
D. B. Wagner

Purpose

The need for an "arsenal of new exterminators" for the "bugs" which have plagued programmers since the earliest days of computing has been thoroughly discussed elsewhere (e.g. see B0021). The collection of programs described here (probe, tracer, breaker, and monitor, BX.10.01-BX.10.04) form an interactive debugging aid which gathers into a very general framework most of the ideas in debugging which have been floating around in separate programs in different systems. This debugging aid is intended for interactive use but will certainly be usable by the batch-oriented user (simply read "control card" for "request" throughout).

A great deal of flexibility is provided through the use of the macro facility (described in BX.1.01) of the command language. One very important feature of this macro facility is that within a macro definition a mixture of commands (lines acted upon by the Shell) and requests (this is the most common word for lines acted upon by individual interactive programs) is possible. Users will not normally communicate directly with the debugging programs but use macros defined in terms of the "bare-bones" requests described in this and the following sections. A collection of "system macros" will be defined, documented, and made available so that the user will not have to know about the full generality of the debugging language unless he wishes to define macros himself.

Notice

A number of points in this and the following four Sections (BX.10.00-BX.10.04) are intentionally vague because at this writing certain parts of the System are not completely "nailed down." This is particularly true of the macro facility. Intentionally vague points are marked with "*" in the margin.

Debugging Facilities

Interrogation: At an interruption or normal termination of a program, the user may interrogate the values of variables and the contents of machine locations; a rather complete expression language makes it possible to conduct these interrogations in terms of the source language of the program. For example if a user, noting some peculiar program output, hits the quit button while a PL/I program is running and types the command

```
probe
```

followed by the request to probe,

```
print a+b
```

he means that probe is to find the storage assigned to the variables a and b in the program, add their values together in the same manner as a compiled PL/I program, and print the result on the console.

Breakpoints: A user may specify that program execution is to be interrupted upon the occurrence of certain (more or less hardware-oriented) events such as control reaching a certain point or a certain amount of time being used up. For example a standard macro named trap could be defined which *
makes arrangements so that the program will be interrupted when control reaches a certain point (label) in the program. (This example is enlarged upon in section BX.10.03) A user would then type .

```
trap sym
```

to cause execution to be interrupted when control reached the statement labelled sym in his program. Then the user would start up his program (probably with a call through the Shell) and wait for the break to occur. When and if it did occur (i.e. when and if control reached sym), he would perhaps type print requests and snoop around in the values of variables at this point in the program's execution exactly as if he had just hit the quit button as discussed above. Finally he might allow execution to be continued, by typing

```
proceed
```

or cause execution to be resumed at some other point, by typing

```
transfer sym2
```

where sym2 is a statement label in the source program.

Tracing: Breakpoints may be used in another way. The tracer Command may be used to store up commands to be executed at specific breakpoints so that what takes place at the break is automatic. A macro named mvar might be defined which *
Causes the value of a variable to be printed every 10 milliseconds. (This macro would contain the command breaker, several requests to breaker, the command tracer, and again several requests. See the enlargement of this example in BX.10.03.) The user could then type,

```
                              mvar a+b
```

start his program by a call through the Shell, and receive the output

```
          a+b                 3.265
          a+b                 3.123
          a+b                 3.145
          a+b                 3.142
```

interspersed of course with any output his program produces.

Process History: One of the actions which can be specified to be performed at breakpoints is that of saving the state of a process so that it can be restored later. One may for example specify that the process state is to be saved every 10 ms. Then for example when and if something goes wrong in the program, *probe* requests can be used to back conditions up in time so that the user can search through time for clues to what went wrong in the program.

Limitations

The debugger is designed to be most convenient to users of PL/I and the standard assembly language. Users of algebraic languages other than PL/I, such as FORTRAN IV, will have to learn some new and occasionally confusing conventions, or else supply a replacement for the expression-evaluating machinery used by the debugging programs. Users of the languages sometimes unkindly called "oddball," such as COMIT, LISP, DYNAMO, ELIZA, and their ilk, will find the debugger as presently conceived less useful, although the trace and breakpoint facilities will probably see some use in connection with these languages. It seems unwise to build in any aids to users of specific special-purpose languages at this time since only an active user of LISP, for example, can have any clear idea of what facilities are useful in debugging LISP programs.

The Programs

Probe (described in BX.10.01) allows the user to examine and modify machine conditions and the contents of his segments using both machine- and PL/I-oriented formats. This is the core of any debugging aid. Considerable experience has been acquired in the matter of machine-oriented formats (e.g., in DDT, FAPDBG, FAPBUG, and GEBUG), but higher-language oriented formats are still in a rather primitive state.

Tracer (described in BX.10.02) provides a convenient tracing facility. In order to use it, the user inserts at strategic points in a program calls to a certain entry in the *tracer* command. Various ways of making these calls occur automatically at specific events will be available, e.g. the breaker and monitor commands and possibly a debug mode in the compiler. The tracer command accepts requests which specify "When argument 1 of the trace call is thus, do this." ("This" may be any sequence of commands and requests to commands.)

The breaker command (described in BX.10.03) accepts requests from a console or macro expansion to place a variety of event breakpoints into a program. It makes arrangements with the System to gain control whenever specified events occur. Breaker amounts to one way of causing trace calls to occur automatically.

The monitor command (described in BX.10.04) accepts requests from a console or macro expansion which indicate that certain blocks of machine code are to be executed interpretively instead of being allowed to run free. Whenever an "execution" access is made to such a block of code, a trap occurs and an interpreter is called. The interpreter calls the trace entry with appropriate arguments after the execution of each machine instruction.

The Debugging Language

An interactive program is an interpreter for a kind of computer language--an "interaction language" rather than a "programming language." The "debugging language" described here uses a number of the conventions of PL/I, e.g., the form of expressions and the control functions if, else, do, and end.

A request is a line which is read and acted upon by one of the programs probe, tracer, breaker, and monitor. (A better word might be primitive, since the requests which are actually seen by the programs will only rarely be typed by the user at his console. As was mentioned earlier, they will normally be used only in macro expansions.) A request consists in general of the request name followed by arguments delimited by blanks. The conventions of the Basic Command Syntax (see BX.1.00) are followed wherever applicable, especially with respect to the "Shell escape character" and the semicolon convention.

Expressions

An expression is something like "a+b" or "sin(a)+6" which can be evaluated to yield a value. Expressions are used in the debugging language in references to variables in the user's program and also wherever numbers, strings, etc. are arguments to requests (as in the specification of loops, see do request, below). Symbols used in these expressions are normally identifiers from the source program associated with the object program under examination. It is absolutely necessary that assemblers and compilers make available to the debugger the details of each compilation: this has traditionally been done with the "symbol table file," a list of the identifiers defined by the programmer in the source program and an indication of "what was done" in implementing that identifier. (The standard format for these symbol tables is described in BD.2.)

A quick description of the debugging expression language would be that it is the PL/I expression language with the values of expressions limited to scalars (a PL/I expression may have a vector or structure value) but with the addition of the data type "address." (The data type "address" may turn out to be identical in implementation to the PL/I data type "pointer", but it seems worthwhile to keep the two concepts separate.) Expressions are divided into two classes, "machine-oriented expressions" and "algebraic expressions." The difference hangs primarily upon whether the "value" of a symbol referred to in the expression is taken to be the *address* (if any) associated with the symbol or the *contents* of the *storage region* (again, if any) associated with the symbol. The values of machine-oriented expressions are not constrained to be addresses, since a "contents" function is part of the language. This function takes an address and returns its contents in the form of a 36-bit bit-string which may then be used in any of the usual ways that bit-strings are used in PL/I expressions.

An *algebraic expression* is any valid PL/I scalar expression in which the variables referred to come from programs written in algebraic languages such as PL/I or FORTRAN IV. The value of a variable is taken to be the contents of the associated storage at the time expression evaluation takes place. If the variable is internal to a (PL/I) block which is not now active, the expression-evaluating machinery attempts to find its value at the last exit from the block. This information may or may not currently exist, depending for instance on the declaration of the variable (e.g. static or automatic) and the strategy used for dynamic storage allocation. The debugger attempts to find a symbol in any of the symbol tables it "knows about." A number of ambiguities present themselves: A name may be used for variables in different separately compiled programs or in different blocks of the same program, and one variable may have more than one generation active (e.g. when a recursive procedure calls itself). To provide a notation for "this symbol in this block," the question-mark (?) is used. For example "a?b" refers to the variable b in the block a. File or segment names may be used in the same way as block names. If a block has no name, its number (counted linearly through the source program) is used instead, so that "a?c?3?b" refers to the variable b in the third block internal to the block c internal to the block a. In the case of a "multiply active" variable (one for which more than one generation exists), the latest generation (representing the deepest recursion) will arbitrarily be used.

A machine-oriented expression is an expression in which the "variables" are symbols from assembled source programs. Here symbols represent either addresses, base-offsets (such as stack symbols), or integers (symbols defined with some analog of the SET pseudo-operation in FAP). Expression syntax remains that of PL/I. In order to allow the expression of complicated Boolean conditions, such as those needed in the specification of searches for machine words with certain content or effective address, several special built-in functions are provided: the "content" function c, the "effective address" function ea, and the Boolean function safe which tells whether it is "safe" to use the effective-address function. This last is made necessary by the fact that in the 645 there are numerous funny kinds of indirection that do not yield proper addresses. The "contents of register" function cr recognizes mnemonics for special registers, so that for example "cr(a)" refers to the contents of the accumulator as a 36-bit string.

The treatment of the dollar-sign ($) in debugging expressions is slightly different from its treatment in PL/I. It is an operator whose preceding operand is a segment name, segment number, or base-register name and whose following operand is an integer giving relative address. The result is of course an address. Thus "alpha$7" means location 7 in the segment named alpha, but "6$7" means location 7 in segment number 6.

"Mixed" expressions, those which include both algebraic identifiers and machine-oriented identifiers, most emphatically do not have an official interpretation. These probably will not cause an error condition but will be interpreted in some reasonably intelligent manner, and may be useful in some contexts; nothing more will be said about these here.

The Control Requests

The four parts of the debugger will recognize, through a common interface, the control requests if, else, do, and end. The request

if condition then request

causes the request to be performed if and only if the conditional expression evaluates true. Then

else request

causes the specified request to be performed if and only if the conditional expression in the last balanced if request evaluated false. The request do (same options as in PL/I)

causes requests following, up to a balanced

end

to be executed under control of the options specified (options are loop-control specifications as in "do j=1 by 1 while a=b;").

In a do specification such as "do j=..." the variable specified is a "pseudo-variable" which is to be specially set up for the purpose. The variable is assigned a data-type consistent with that of the value of the expression to which it is being set, storage is assigned, and the variable name is placed in the symbol table. When the range of the do is left, the storage is freed and the name removed from the symbol table.

In addition to the above requests, each of the four parts of the debugger recognizes the request

exit

which means to return to the calling program, normally the Shell.

Published: 04/09/69
(Supersedes: BX.10.00A, 01/06/69,
BX.10.00A, 07/18/68)

Identification

Interactive Debugging Aids for Initial Multics
M. Wantman

Discussion

MSPM section BX.10.00 describes an elaborate system of debugging aids for Multics. While that section describes the system envisioned for the ultimate versions of Multics, it was thought desirable to have a simplified debugging system which could be used with Initial Multics. This interim version is expandable and eventually should include all the features described in BX.10.00.

Facilities

The interim debugging system provides ways for a user to obtain the following types of information:

a) machine conditions and register contents

b) segment names and numbers and access information

c) selective dumps in octal of a segment between user-specified limits.

d) dump contents of an entire process directory

e) forward or backward stack traces

f) argument lists.

g) list of available requests

In addition, the user may make changes which affect the subsequent execution of the process. These are

h) make a segment known or unknown to the process

i) make an octal patch to a segment

No provision has been made for symbolic addressing of locations within segments, or for changing the contents of registers or segment locations. These, as well as other features described in BX.10.00, will be implemented in stages, and BX.10.00A will be updated periodically.

Usage

Entry to the debugging system is accomplished through the procedure "probe". Requests for specific information are then given to "probe", which passes them on to the appropriate procedures. "probe" is invoked in one of two ways:

1) from command level. This can be done after a process incurs an unexpected signal and "unclaimed_signal" has called the listener. It can also be done after the user quits a process that has been running and the quit responder is invoked.

2) from a running process. This occurs if a call to probe is encountered in a procedure segment. Once the call to probe has been executed, the user can "snoop" around at his leisure. Execution of the original procedure can be continued by simply leaving probe via the "quit" request.

Once probe has been entered, it handles all user input lines until the "quit" request is given.

The following requests are recognized:

arglist	print argument list for one stack frame
dump_process	print entire contents of a process directory
info	print a list of requests available
initiate	make a segment known to the process
output	change routing of output
quit	leave probe
segdump	dump segment in octal
seginfo	list segment status information
set	place information in write-permit segment
stack	print stack trace
state	obtain machine conditions
terminate	make a segment unknown

arglist

The format of this request is

```
arglist s1 s2
```

It prints the values of arguments which were passed to the stack frame designated by s2 in the segment s1. s1 is the name or number of the stack segment, and s2 is either the location in octal of the beginning of the stack frame, or the name of a segment. If it is a segment name, the stack is searched backwards for the latest occurrence of a frame used by the segment. If no appropriate frame is found, a diagnostic message is printed.

If only one argument is given to arglist, the argument is interpreted as a location within the current stack segment. That is,

```
arglist seg_name
```

is interpreted as

```
arglist stack_xx seg_name
```

dump process

Request format is

```
dump_process   -id-
```

where id is the unique id of the process expressed in octal or as a character string. If it is not present, the user's process is assumed. The process directory is scanned and each segment is dumped in octal. Normally output will be directed to a segment for later printing.

info

Execution of this request will cause printing of a short description of the requests accepted by probe. The message will refer the user to this document for more complete information.

initiate

Makes a segment known to the process and assigns a segment number. The format of the request is

```
initiate path -callname-
```

The segment found at path in the file system is initiated with call name callname. If callname is not specified, the entry name of the segment will be used. If the segment is successfully made known or is already known, the comment

segment (path) initiated with call name (callname). Number (n)

is printed. If it could not be initiated, the comment

    segment (path) not initiated

is printed.

output

All output from probe is directed initially to the user's console. It may be desired to have output go to a segment and have the segment printed off-line. The request

    output segment s

will direct output to the segment s. If no segment by that name exists, one will be created with access RWA. The comment

    output directed to segment s. Number (n)

will be printed on the console. If the segment is filled to the 64K limit, its bit count is set and another uniquely-named segment is started.

Output will be redirected back to the console by the request

    output console

The segment that was being written will have its bit count set. If output is directed to the segment again, the later information will be appended to the end of the segment.

quit

When this request is given probe returns to command level. If output had been directed to a segment, the bit count on the branch is set.

segdump

This request is used to dump all or part of a segment in octal. The segment must be known to the process, and may be specified by name or number. The format of the request is

    segdump s -lbound- -hbound-s is the name or number of the segment to be dumped, and the lower and upper bounds are given by lbound and hbound respectively. If the lower and upper bounds are not specified, the entire segment is dumped. If the upper bound is not specified the segment is dumped from n1 to the current length as defined by the file system.

If the segment is not known to the process, the comment

```
segment s not yet initiated
```

is printed. The segment may be made known by the initiate request. If the user does not have read access, the information is copied via ring-0-peek into the users stack and printed.

<u>seginfo</u>

This request prints a list of names and numbers of segments known to the process. The format is

```
seginfo s1 s2 -all- -long-
```

s1 and s2 are segment names or numbers in octal, and indicate the range of segment numbers for which the information should be printed. For example, the request

```
seginfo 200 test_proc
```

will print the names and numbers of all segments whose numbers lie between 200 and the segment number of test_proc. If neither s1 nor s2 is specified, the information is printed for all segments. If only s1 is specified, the information is printed for s1 only. If the lower bound is higher than the upper bound, a comment is printed.

The presence of the optional parameter "all" in the request will cause all the call names by which a segment is known to be printed. The presence of the parameter "long" will cause printing of the current length of each segment, its access attributes, and its date of creation.

set

This request allows the user to make changes to any segment in his process for which he has write permission. The format of the request is

```
set s|n v1 -v2- -v3- ...
```

S is the name or number of the segment to be altered, and n is the location where the first new value is to go. Blanks may or may not be present on either side of the vertical bar. The status of segment s is examined to determine if the user has write access. If he does not, a message is printed and no patching is attempted.

Otherwise the values represented by v1(v2 v3 ...) are placed in locations n(n+1,n+2,...) of segment s. A message is printed containing both the old and new values to minimize the probability of error and to facilitate restoration of the original values.

stack

Multics keeps a partial history of the course taken by a process. The "stack" request makes some of that information available. It prints the name and number of the procedure using the frame, the starting location of the frame, and its size. The format of the request is

```
stack -s1- -s2- -f- -args-
```

where all parameters are optional.

s1 is the stack segment to be examined, s2 is the location in the stack where tracing is to begin, "f" indicates that the trace is to proceed forward in the stack, and "args" asks for an argument list (as in the "arglist" request) to be printed for each stack frame.

s1 may be specified by name or number. If it is not given, the current stack segment is traced from the end to the beginning. s2 may be specified only if s1 is given. If s2 is given as an octal number, it is interpreted as the starting location of a stack frame. If s2 is the name of a segment, the stack is examined from the end for a frame belonging to s2. If one is found, tracing begins at the frame. If none is found, a message is printed for the user.

The forward option "f" can be specified only if s1 and s2 are given. If it is present, the trace proceeds from s2 to the current end of the stack. The option "args" can occur anywhere in the request. If it is present, "stack" will print a list of all arguments passed to each stack frame.

state

The "state" request prints machine conditions as requested by the user. It searches backward through the current stack trying to find an occurrence of a frame belonging to "signal". If one is found, and it is preceded immediately by an occurrence of the FIM (fault interrupt module), then the FIM frame contains the machine condtions at the instant the fault occurred.

If state receives a number as an argument, it prints the machine conditions existing at that location in the current stack segment. If no signal-FIM combination can be found, register contents are extracted from the stack frame of the procedure that called probe.

The following parameters are recognized by state:

arith	print A, Q, and exponent registers
timer	print timer register
location	print location of fault and the effective address
index	print 8 index registers
bases	print 8 base registers
cunit	print control unit and ring number

If no parameters are given, all the above information is printed.

terminate

This request makes a segment unknown by removing it from the KST (known segment table) . The format is

```
terminate s1
```

S1 is the pathname of the segment to be terminated. If no directory is specified the current working directory is assumed.

(Note: When a segment is made unknown, its linkage section is not removed from the combined linkage section. If a different version of the segment is later made known, you will continue to work with the older linkage section. This can lead to unpredictable results).

Summary of requests

arglist	-s1- s2
dump_process	id
info	
initiate	path -callname-
output	medium -name-
quit	
segdump	s1 -lbound- -hbound-
seginfo	-s1- -s2- -all- -long-
set	s1|n v1 -v2- -v3-
stack	-stackseg- -s1- -f- -args-
state	-loc-
terminate	callname

Implementation

Entry to the set of debugging commands is achieved by calling the procedure probe with no arguments. Probe then calls dispatch_request with no arguments for each new request line. After completion of each request probe prints the line.

-

to indicate that the user should enter his next request.

The procedure dispatch_request calls read_in to get the request line and hands it to the debugging parser via the parse_scan$prime entry. It then calls parse_scan$atom to get the first group of characters. This should be the name of one of the basic requests. A table of request names and corresponding procedures to call is kept in the eplbsa segment debug_data. Dispatch_request searches this table and calls debug_data with the index to the request table, which makes the call to the appropriate procedure.

Each request makes calls to parse_scan to get its arguments and interprets them appropriately. If a request finds that it cannot do what is asked of it, it prints an error comment and returns to dispatch_request which returns to probe which waits for the next request.

If the search made by dispatch_request fails, the request is passed directly to the shell, which treats it as a normal command line. Thus any Multics command can be given without leaving probe.

For details of the implementation of the individual requests, see MSPM sections BX.10.05-BX.10.20. For abstracts, see MSPM section BS.

Published: 6/7/66

Identification

Machine- and PL/I-oriented interrogation and modification of the contents of segments
Probe
D. B. Wagner

Purpose

Probe is an interactive program which allows the user to peek into and modify the segments of a process at any interruption of the process or after a normal termination. It reads its requests through a common interface as mentioned in BX.10.00, so that for example the if and do Requests facilitate the definition of macros which perform various kinds of searches.

Usage

The command

```
probe
```

causes probe to begin reading requests from the console. The user may type any of the requests listed below or any of the "control" requests (if, else, do, and end) described in BX.10.00. He may also type macro invocations (in the same form as in the command language: see BX.1.01) which expand to sequences of these requests. If a line received by probe (after macro expansion) is not recognizable as a request, it is treated as a command. The line is given to the Shell, which gives an appropriate diagnostic if it is not a command either. *

Interrogation Requests

One request, backed up by numerous special functions built into the expression-evaluating machinery, provides the basic interrogation facility. This is

print *expression* *expression* ...

The values of the expressions are printed on one line on the console, separated by tab characters. Each expression is normally an invocation of one of the built-in "format functions." A format function takes some argument and returns a character-string which "represents" the value of that argument according to some interpretation. (See the example below.)

The format functions are listed below. Operation of most is obvious. Each takes a PL/I scalar or an address and returns a character string which is suitable as a representation of the value of the scalar according to some interpretation.

decimal
floating
octal
binary
ascii
instruction
indirect
symbaddr

Most of these are from GEBUG, and behave in essentially the same way as the corresponding GEBUG output formats. The instruction format function takes an address and produces a representation of the contents of that address which looks like a line from an assembly, i.e., something like "lda sym,4". If a symbol table is available for the segment involved the address is printed symbolically. (One of the lessons of FAPDBG and GEBUG is that this sort of symbolic instruction printing involves a great number of aesthetic problems which can be only partially solved. A number of known bugs exist in FAPDBG and GEBUG, such as the one which causes printing of "ALS 11" as "ALS SYMBOL-4763", but usefulness is only slightly impaired by such nonsense.) Indirect again takes an address and interprets its contents as an indirect word. An optional second argument specifies the type of indirect modifier the instruction referring to the indirect word would have (e.g. *, SC, CI, etc.--modifier mnemonics from the assembler). Symbaddr takes an address and produces a representation of the address in the form "segnam$expression", where the expression is in terms of whatever symbol tables may be available for the segment.

A macro (defined to perform the same operation as the GEBUG "peek" request), invoked to print the contents of locations * 69 through 105 of segment alpha interpreted as decimal and octal, might produce the following sequence of requests:

```
do q=alpha$69 by 1 to alpha$105
print symbaddr(q) decimal(c(q)) octal(c(q))
end
```

A "search" macro invoked to search the same area for a word with bit 29 on might expand to the following:

```
do q=alpha$69 by 1 to alpha$105
if substr(c(q),29+1,1)="1"b then print symbaddr(q)
end
```

Modification Requests

A user may wish to alter the contents of his segments at an interruption of the execution of a program for either of two reasons: he may wish to correct a bug without retranslating (despite the fact that "patching" will be less desirable in Multics than it has been in other systems), or he may be trying to answer such a question as "What would be the effective address of this if index register 2 contained that?" or "Would this complicated PL/I conditional expression evaluate true if alpha were 25.7?"

To allow both of these uses, with emphasis on the second, the requests *set*, *reset*, *revert*, and *fix* are provided. The request

set *variable*=*expression*

saves the current value of the variable in a stack and alters the value to that of the expression. "The value of the variable" means, for a symbol used in an assembly, the address it represents (the value of the expression must be an address), and for an algebraic variable the contents of the storage region associated with the variable. To patch machine locations, the format

set c(*address*)=*expression*

is used, meaning set the contents of the location.

A patch normally remains in force only until *probe* is left, either by a normal return (*exit* request) or by a transfer into the program (*transfer* or *proceed* request), at which point the saved value is restored. The following requests modify the application of this rule:

```
fix variable
fix c(address)
reset variable
reset c(address)
revert variable
revert c(address)
```

Fix makes the patch permanent, so that there will be no automatic resetting of the value. *Reset* changes the value of the symbol or the contents of the location specified back to its value the last time *probe* was entered (the value at the bottom of the stack). *Revert* changes it to the next previous value saved by the *set* request (the value at the top of the previous-values stack).

Handling the Trace File

A trace file may be used to keep a history of a process. This is a ring file of size determined either by default or by specific declaration into which the *snap* request described below stores snapshots of interesting storage regions. Then the *forward* and *backward* requests allow the user to roll machine conditions back and forth in time (loosely speaking: see the discussion below) so that the full power of *probe* may be used to peek at conditions at various times in the history of the process. The trace file will also see considerable use in conjunction with a "time control" attached to displays analogous to the Scheduling Algorithm Display in CTSS and for statistics-gathering.

Normally the following request will be used only through the *tracer* command.

snap *region*, *region*, ...

Each region specification takes either of the forms

variable
address to *address*

where the symbol is an algebraic variable and the expressions are addresses. This request causes a snapshot of the specified storage region to be placed into the trace file. The request

snapall

takes a snapshot of every impure (changeable) segment attached to the process. (Or rather, a snapshot of enough information so that the state of the process can be brought back to this instant in its history. Clearly there are problems concerning the parts of the process which are inaccessible to the user.) The amount of information stored can easily become rather gross and this request must be used with some care. The usefulness of this sort of all-inclusive snapshot, however, is clear: It makes possible a "time-machine" which allows the user to jump freely about in history and to experiment with changes in the properties of the primordial ooze from which a set of machine conditions sprang. The requests

forward
backward

roll machine conditions forward or backward in the trace file. That is, they read the next and previous snapshot respectively and *set* (see above) the contents of the corresponding storage regions accordingly. It is to be noted that this can lead to an inconsistent muddle if not enough information is saved in the trace file, e.g. if *snap* requests are used to do the saving instead of *snapall*. The trace file is a tool whose use is not always appropriate.

A "time-search" macro, invoked to find an instant in the history of the process when the variables a and b in some PL/I program were equal, might expand to the following sequence of requests (note that "¢n" is the escape for the negation sign):

```
do while a¢n=b
backward
end
```

Miscellaneous Pequests

The request

proceed

causes program execution to be resumed at the point at which it was last interrupted, either by a quit or by a breakpoint or other call to the trace entry (see BX.10.02-03). This *
involves a "synthetic epilogue," described in detail in EPL design journal #4 (B0005), which deactivates active blocks of the debugging programs which are above the program in the stack, as in a PL/I "non-local go to."

The request

transfer *address*

performs a synthetic epilogue and transfers control to the *
location specified.

The request

exit

causes *probe* to return to its caller, normally the Shell.

Published: 6/7/66

Identification

Program tracing under interactive control
tracer
D. B. Wagner

Purpose

Use of tracer allows the execution of a program to be monitored on-line in as fine or coarse a manner as desired.

Usage

To use tracer the user inserts at strategic points in his program calls (using the standard call sequence) to the entry tracer$report in the tracer command. Some ways of causing these calls to occur automatically on occurrence of certain events will be provided, such as the breaker and monitor commands described in BX.10.03 and BX.10.04. There may also be a debug mode in the PL/I compiler which causes such calls to be inserted between statements, giving full information concerning variables changed, previous and new values, etc. The tracer command is then used to store up actions to be performed whenever tracer$report is called with certain arguments. These actions may include both commands and requests to commands.

The format of the calls to tracer$report is essentially up to the user, except that the first argument must be a character-string name for the call which will be used to identify the actions to be performed.

The command

tracer

causes tracer to begin reading requests from the console. The user may type any of the requests listed below or any of the "control" requests (if, else, do, end) described in BX.10.00. He may also type macro invocations (in the same form as in the command language: see BX. 1.01) which expand to sequences of these requests. If a line received by tracer (after macro expansion) is not recognizable as a request, it is treated as a command. The line is given to the Shell, which gives an appropriate diagnostic if it is not a command either. *

*

Implementation

The following digression is necessary to explain the action of the tracer command. See the diagram of Fig. 1. The command listener, the debugging programs, and most other

Interactive programs read their requests through a "request handler" which acts as an interface to the I/O system. It expands macros, handles the semicolon convention, etc. The request handler keeps a special data base called the request queue, and before the request handler reads a line from the console it checks to see if there are any lines waiting in the request queue. If there are it uses the first line in the queue instead. (The macro processor will be one program which places lines into the request queue: the entire first-level expansion of a macro invocation is simply put at the head of the queue.)

When the setaction request described below specifies a name and a list of commands and requests, these are stored in the tracer data base. When eventually the user's program is started and a call to tracer$report occurs, the first argument of the call is matched against all the stored-up names. If a match is found the corresponding list of command and request lines is placed at the head of the request queue and the command listener is called. This scheme provides a very general tracing facility.

Requests to Tracer

Setaction and endaction are the basic requests: the sequence

```
          setaction name
          .
          .  action
          .
          endaction
```

causes the action specified to be stored away in a data base used by the trace entry. The name is a character-string identifier, to be matched against the first argument of each trace call. Action is a sequence of commands and requests. It is stored up to be performed whenever a call to the trace entry tracer$report has the first argument equal to name. If more than one action has been specified by setaction requests for a given name, they will be performed in the order given. Name may be "*", in which case action is to be performed on every call to tracer$report.

The action specification may of course include conditional (if ... then ...) requests which narrow down the selection of action still further than the naming convention does. Expressions may include the special function

tracearg(n)

which gets the n'th argument of the last trace call.

The request

resetaction name

deletes all actions stored up for the name given.

The requests

listaction
listaction name

cause either all currently specified actions, or all currently specified actions for the given call name, to be listed in a convenient format.

The request

exit

causes *tracer* to return to its caller, usually the Shell.

Identification Published: 6/7/66

Breakpoint processor
breaker
D. B. Wagner

Purpose

Breaker accepts requests to interrupt the execution of a program upon the occurrence of certain events. The tracer command (see BX.10.02) is normally used to specify actions to be performed at each break.

Usage

The command

breaker

causes breaker to begin reading requests from the console. The user may type any of the requests listed below or any of the "control" requests (if, else, do, end) described in BX.10.00. He may also type macro invocations (in the same form as in the command language: see BX.1.01) which expand to sequences of these requests. If a line received by breaker (after macro expansion) is not recognizable as a request, it is treated as a command. The line is given to the Shell, which gives an appropriate diagnostic if it is not a command either. *

Requests to Breaker

The request

setbreak name event

causes arrangements to be made with the System so that when the specified event (see below) occurs breaker will regain control and make a call to the tracer entry tracer$report, then allow the program to resume running. Name is a character-string expression (see the discussion of expressions in BX.10.00) which is to be used as the first (identification) argument in these calls. Event is one of the following (meanings are usually clear: detailed explanations are avoided here to keep this document to a manageable size)

extime n ms
extime n sec
extime n min
extime n hrs

realtime (same arguments as extime)

```
access spec location to location
access spec variable
     (spec= a combination of X (execution), R
     (read), W (write))

call from seg
call from seg$expression
call to seg
call to seg$expression
call from ... to ...

return (same arguments as call)

extref (same arguments as call)
```

Call and *return* refer to subroutine calls and returns (as in the CTSS command STRACE). These breaks and the *extref* (external reference) break are implemented using special entries to the Linker similar to those described in BE.12.01 for the 645 simulator system.

The *access* event break is implemented by temporarily changing the descriptor bits for the segment involved and arranging to have access-violations reflected to the debugger's trap-handling routines. This method can of course be rather costly, especially if a small block out of a large segment is specified, since all accesses of the specified type anywhere in the segment must be executed interpretively. It is expected that interpretive execution of instructions will take an average of 50x normal execution time.

There is a problem with the *extime* (execution time) event: within the system execution times are measured in core cycles, not in conventional time units. Hence in addition to the time units mentioned above for the *extime* request (which have to be approximated in terms of core cycles), the units *kc*, *mc*, and *gc* (kilocycles, megacycles, and gigacycles) are provided.

The request

exit

causes *breaker* to return to its caller, normally the Shell.

Examples

A "watch" macro might be defined to permit the monitoring of the values of variables through time. When invoked to watch the variable *beta* in a PL/I program and report every 10 ms., the macro might expand to the following sequence, which as can be seen includes both commands and requests:

```
breaker                        (command)
setbreak "xyz" extime 10 ms    (request)
exit                               "
tracer                         (command)
setaction "xyz"                (request)
probe                          (command to be stored)
print "beta" beta              (request to be stored)
proceed                            "
endaction                      (request)
exit                               "
```

When the program is started up, the following lines, interspersed of course with normal program output, might be typed on the console:

```
beta        6.723
beta        8.927
beta        5.400
beta        6.723
beta        8.927
```

and this might or might not give the user a clue to what is going wrong with his program.

A macro to perform the same function as the "B" request in FAPDBG (break when control passes to a specified location and begin reading requests) might expand to:

```
breaker                         (command)
setbreak "xyz" access X location
                                (request)
exit                                "
tracer                          (command)
setaction "xyz"                 (request)
probe                           (command to be stored)
print "BREAK"                   (request to be stored)
endaction                       (request)
exit                                "
```

When and if control reaches the location specified, the command <u>probe</u> will be called. It will print "BREAK" and begin reading requests. When the user eventually types the <u>proceed</u> request, the program will start running again.

Published: 6/7/66

Identification

Instruction-by-instruction interpretive execution of programs.
monitor
D. B. Wagner

Purpose

Monitor accepts requests which cause certain areas of programs, whenever entered, to be executed interpretively instead of being allowed to run freely. Used in conjunction with the tracer command it allows very tight control of the execution of a program which is causing trouble.

Usage

The command

monitor

causes monitor to begin reading requests from the console. The user may type any of the requests listed below or any of the "control" requests (if, else, do, end) described in BX.10.00. He might also type macro invocations (in the same form as in the command language: see BX.1.01) which expand to sequences of these requests. If a line received by monitor (after macro expansion) is not recognizable as a request, it is treated as a command. The line is given to the Shell, which gives an appropriate diagnostic if it is not a command either. *

Requests to Monitor

The request

setmon location to location

causes arrangements to be made so that all execution of instructions in the block specified is done under strict supervision. Whenever control reaches a location in a block specified in a setmon request, an interpreter gains control of the process (by the same mechanism as that used by the access event in breaker) and executes the machine instructions interpretively. At every instruction a call to the tracer entry tracer$report (see BX.10.02) is made with appropriate arguments. Name is a character-string expression which is to be used as the first (identification) argument in these calls. It is expected that interpretive execution of machine instructions will take roughly 50x normal execution time.

The request

resetmon name

resets any setmon's which have been given for the specified name.

The request

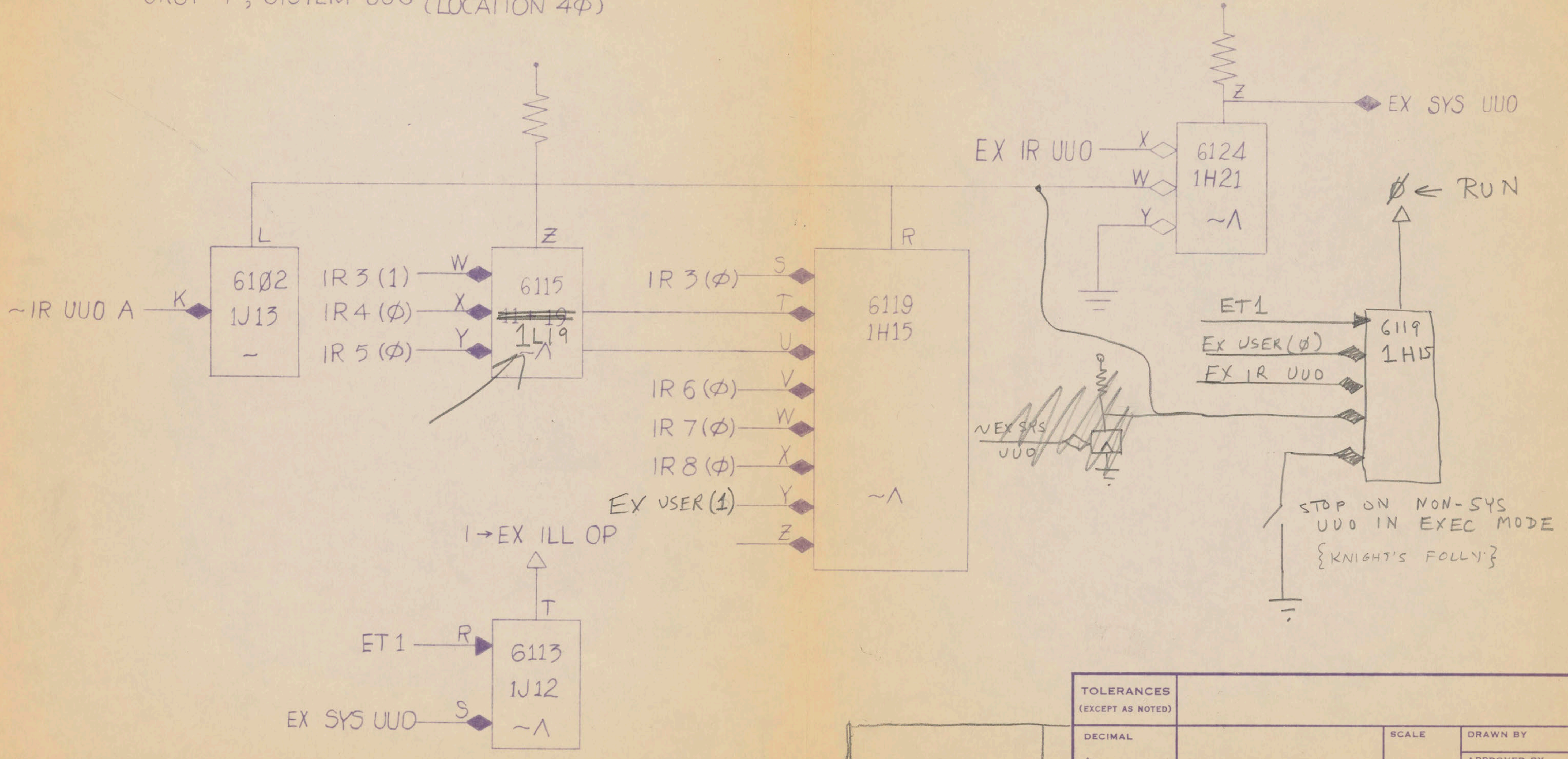
exit

causes monitor to return to its caller, normally the Shell.

DATE	SYM	REVISION RECORD	AUTH.	DR.	CK.

UUO Ø V
UUO 4Ø ~ 47 V
ILLEGAL IOT V
JRST 4 ; SYSTEM UUO {TRAP TO LOCATION 4Ø}

UUO 1-37 V
UUO 50-77 ; USER UUO {TRAP TO 40 + C(RELOC)}



TOLERANCES (EXCEPT AS NOTED)			
DECIMAL ±		SCALE	DRAWN BY APPROVED BY
FRACTIONAL ±	TITLE USER MODE UUO MODIFICATION		
ANGULAR ±	DATE	DRAWING NUMBER	

PDP1Ø WORKING COMMITTEE

Here is a copy of the overrated report of the first meeting of the working group. It is noteworthy for its omissions. In particular, the ideas and criticisms of Carl Ellison and Bill Wulf are poorly represented, and the discussions of the features of the MAC and BBN systems are wholly insufficient (we need more specifics about the BBN system). Hopefully the meeting on 26 January will polish up these difficulties.

I will be in touch on the exact time of the meeting, but if you don't hear, assume 9:ØØ AM at BBN.

PEACE. Bob Sproull

This rather sketchy document represents some of the considerations discussed at the first meeting of the ARPA Users' Working Group on 19 December 1969. This is highly preliminary, and will need heavy revision.

I. Why cooperate at all?

This question is perhaps not as silly as it may at first seem. One of the considerations necessary to specify a time-sharing system for a community of users is an adequate characterization of those users' habits and their requirements. There are several requirements of the ARPA Users (hereafter referred to as the AU's) which suggest that they might want similar time-sharing systems at their installations. First, many of the users of the DEC 10/50 system feel that it leaves something to be desired. It should be remembered that the system was designed and built in what are now the dark ages, and that it has served rather well for the original purposes. However, the AU's research-oriented work often depends on the versatility of the system supporting that work, and much of that work finds the DEC system somewhat recalcitrant.

Second, the AU's may benefit from compatible systems at all PDP10 installations. Programs generated by one outfit can then be used by all. If the ARPA net becomes a useful thing, these systems can enjoy identical ways of servicing network messages. The cost of implementing a new compiler or a new system feature at all installations will thus be rather low.

Third, given that the AU's will have to do some system development to get any time-sharing system tuned to their special requirements, there is a potential saving in manpower and time by cooperating in the effort, or investing some particular group with responsibility for the task.

II. What possibilities exist?

We envision about three ways of tackling the problem of providing ourselves a system: (1) modify the DEC system to include the shiny new features we all feel we need, (2) adopt and modify the MAC or BBN systems, and (3) write a new system from scratch.

Our views on the feasibility of these various paths are reserved until after some discussion of the requirements of the system. This following discussion is rather inutile in that it does not constitute a concrete specification of a time-sharing system that could be built. It is rather a list of prejudices -- areas or ideas that should be emphasized in the system we would want. It is also a list of problems, e.g. providing real-time service. The discussion is merely to suggest that the system designers or modifiers should tackle these problems, and seek an implementation consistent with other aims of the system.

In short, we did not feel ourselves sufficiently adept system-designers to list imperatives, and resorted rather to woozing around in that beautiful meadow of indecision.

III. General performance goals.

We would like to try to characterize the user environment for this time-sharing system. We expect that most installations will want to service about 3Ø users simultaneously from as many terminals. Some installations may have a need for batch-like processing abilities as well. Of course, the exact quality of this service is impossible to specify. It will depend heavily upon particular I/O and file system speeds, as well as the job mix running in the machine.

Several of the existing PDP1Ø groups (notably Stanford) have had experience with time-sharing for AU-like users. It is not easy to cite many specific recommendations emanating from this experience. We can, however, say something about the job mixes observed, and the reactions of users to various kinds of service. Stanford finds typically three kinds of users:

1. The editors. A small editing program (now 9 K) is used by a great quantity of people for long periods of time. The editor makes use of control characters and special echoing to do intra-line editing. It is thus highly desirable that the editor be given service sufficient to do the special echoing as fast as possible. Users are very sensitive to the response time when editing a line. If the editor must go off and do some complicated string search or a long file operation, users are very understanding of the delays, since they realize a fair amount of computation may be required.

2. The assemblers and compilers. The next portion of the debug loop requires assembling and loading your program. These processes are have several unique characteristics: they make heavy demands on the file system, require essentially no user interaction, and usually can make reasonably good estimates of the amount of core

and compute time required to accomplish the appointed task. Typically, the assembler or compiler will run in 30 K for a large program, and consume 15 seconds to 1 minute of compute time.

3. The debuggers. People almost never RUN their programs, but instead debug them. For examining locations in their address space (which may require searching some symbol table), the user likes fast response. When he asks that a portion of his program be executed, he will tolerate some delay. These jobs are typically 30-86 K long.

The scheduling system which developed from these requirements attempts to give a uniform level of service to all users. Statistics are kept about the percentage of run time requested that is actually granted. These statistics have a time constant of about 20 secs. Hence, if a user suddenly becomes runable because a user interaction has terminated, he will very often run until the process is blocked for more interaction. This is better service than merely putting jobs emerging from teletype wait in a high priority queue.

BBN reports the use of their system to be typically 10 editors and 10 large LISP users. Clearly, the system they are building is designed to service a much larger number of jobs.

IV. Hardware requirements.

We suggest that there is a minimum hardware configuration which is suitable for the time-sharing system we propose. The system should be able to function adequately in this minimum configuration, and should also be able to benefit from (certain) augmentations of that configuration.

First, we assume one (1) PDP10 processor or equivalent (enter the IC 9000). It should have all the hardware options (i.e. a 10/50 -- byte instructions, floating point, although not necessarily protect-relocate).

Second, it seems unreasonable to develop a time-sharing system which cannot count on at least 128 K core. This is not an absolute requirement, but the whole issue of process switching revolves around such questions as amount of core, swapping speed, latency, etc.

Third, and more long-windedly, a memory map is required. Some of the inadequacies of the DEC system result from the terribly simple memory access. The motivations for the map are many, and this is no place to try to summarize the last decade of CPU designs. The AU's stand to profit chiefly from two aspects of a map:

1. The ability to share code and data. This ability is entirely lacking in the DEC system. Its lack hurts both because of lost efficency (larger core images since code cannot be shared) and lost features (the only way of sharing data is with the second protect relocate mechanism, which is an underdeveloped resource in that the DEC cusps cannot load code in the second segment in a nice way).

2. The ability to create monstrous amounts of active storage

even though this storage may be accessed sparsely (e.g. LISP and LEAP). This represents a class of problem solutions which are not feasible under time- sharing on the DEC system.

We suspect that a memory map is not a costly piece of hardware. If the installation already has a CPU, 128 K core and some secondary storage medium, there is at the very least $400,000 invested already, compared to the (probable) 30-50 K cost of a mapping box.

We believe that a single-level paging map is most useful to the AU's. A full segmentation system can potentially provide a far richer virtual machine for the user, but has several disadvantages:

1. The PDP10 address field is not really organized properly for segmentation. This is due partly to its length (only 18 bits) and partly to the way indexing arithmetic is performed.

2. Implementation of a system designed to make full use of the segmentation may require extensive experimentation with software. It is our belief that the AU's consider this system to be a service, and should have the attributes of such (infinite reliability, good documentation, high versatility). This system should not be an experimental one.

3. Unless some cleverness were used in the design of the segmentation map, the system could not possibly run programs written for a standard PDP10.

4. Some of the usefullness of segmentation can be approached in a paged environment by giving the user control over the

disposition of his address space.

5. It has not yet been demonstrated to our satisfaction that the benefits of segmentation exceed the cost of segmentation.

If we can agree we all want a paging box, can we agree on the page size? We could not. Ideally, we envision a piece of hardware which is based on a 9 bit page key and a 9 bit line number, together with a system which is capable of any effective page size. Thus if a particular swapping device has characteristics which suggest using a 1024 word page size, then the system will allocate and manage pages of that size. The mapping box, however, is pointed at data which look like 512 word pages, but are diddled appropriately. Unfortunately, this is a relatively difficult software feature to provide.

BBN has thought fairly carefully about the optimum page size for their applications. They arrived at a 512 word page. This is based on several features -- first that a full page table occupies exactly one page, and second that BBN's heavy use of LISP suggests a large number of relatively sparsely populated pages. The possible cost of such a small page size will result from those operations which are performed on a page-by-page basis, such as swapping, allocating and seeking to the appropriate disk track, core allocation, etc. BBN is convinced that the most serious problem here is in the time to process a swap request, and that sufficient care has been taken to minimize the problems there.

The MAC mapping box is designed around a more standard 1024 word page. Both the BBN and the MAC hardware designs are reasonably easily altered to change page size. The more serious inflexibilities would be in the system, particularly in such matters as allocation of tables for each process, etc.

The addition of a memory map cannot be accomplished merely by attaching a box to the PDP1Ø memory bus. There are some changes required in the PDP1Ø CPU in order to use the map effectively. Most obviously, the system wants to be able to use the user map occasionally for referencing the user's address space. The most flexible implementation of this is to install an XCT mapped feature. The four accumulator field bits of the XCT instruction can then be decoded to mean:

--map any indirection references & BLT final address

--map normal fetch, store, fetch byte pointer, "from" address of BLT.

--map indirection specified by byte pointer,"TO" address of BLT.

--map fetch and store on LDB,DPB. This particular allocation of bits and functions is BBN's. There are a couple of specific problems -- suppose the instruction executed refers to a user address below octal 2Ø, i.e. an accumulator -- do you always want to map that to the user's shadow core?

Other hardware modifications are equally imperative. The mapping box wants to trap when an attempt is made to store into write-protected pages. On the PDP1Ø, the memory store is accomplished after the AC store. Hence, it is necessary to either

(1) remember the data which was about to be written into the protected page, and then trap (BBN), or (2) rearrange the store cycles on the PDP10 so that the memory store happens first (MAC) -- thus the whole instruction may be executed again.

The modifications described above have been implemented by either MAC or BBN and have not overflowed the very small amount of spare card space on the PDP10.

BBN has implemented a particularly nice BRS-(940)-like instruction. We mention it here at some length since almost any system might enjoy such an instruction. The instruction is called JSYS. The effective address of the JSYS is computed. If the effective address is less that octal 1000, a dispatch entry is located at (absolute) 1000 + E. The left half of the dispatch word specifies an address for saving the user mode PC, and the right half specifies an entry point. The PC is switched to this entry point, in exec mode. A jumper card specifies whether there are 64, 128 or 512 words of dispatch table entries. If E is greater than 1000, the same happens except the dispatch table is located in user address space, and the PC remains in user mode. The main advantage of the JSYS over the UUO is that no decoding is necessary to discover what function the user wants. Thus some system calls may be accomplished with blinding speed (such as byte-at-a-time file I/O).

We consider it imperative that the system have a large amount of secondary storage. This may be divided into separate hardware units for efficiency (e.g. swapper and file system). A time-sharing system is really a somewhat special file system, and thus must have the storage necessary for keeping these files. Note that this says nothing about swapping per se. If you don't want to swap actively addressed pages onto the secondary storage medium (e.g DECTAPE), fine.

One of the most important aspects of a time-sharing system is the terminal used by the user, and the way the system appears to the user at the terminal (see discussion of execs). Unfortunately, there is no hope of standardizing on any one kind of keyboard device. We consider it old-fashioned to cater to half-duplex lines, but some users (e.g 2741's) may require that service.

Much of this discussion will be deferred until considerations of software, but something must be said about character sets. The terminal is one of the few places that an association is made between some bits and a graphic. We could not reach agreement on a particular character set, but 96 character ASCII seems to be the minimum permissible. Stanford will not part with its 128 character modified ASCII plus control bits (10 bit characters maximum) that can be typed in. There are other motivations for standardizing on character sets -- if any sharing of programs written at different installations is to be useful, we will have to agree on character set conventions.

We suggest as a guide that a user should be able to see some major fraction of the system from a model 33 teletype. There may be some internal characters that the system cannot type out at him, but these should not be essential to his understanding of what is happening.

Since we are currently discussing hardware configurations, let us diverge from the minimum configuration to one of the possible augmentations of this minimum system -- a two processor system. As the compute load at an installation grows, it would be convenient if the addition of another processor would help alleviate that load, and still allow users access to all the memory and I/O gear of the original processor (file system, terminals, special devices). The BBN system is being designed for dual processors. They are making the code maximally reentrant, and at the same time making it easy to remove the code required to run two processors (critical section locks, etc.), Clearly the second processor must be identical to the first -- all CPU modifications must be identical; it must have another identical mapping box.

The dual processor system would entail at least one and probably two additional pieces of hardware. The first is a hardware link between the processors to allow one to interrupt the other and vice versa. The second device is an I/O bus multiplexor. BBN is not planning to build such a device, but instead route I/O requests to the processor with the device attached to its I/O bus. MAC has designed a very clever I/O bus multiplexor which essentially "attaches" each hardware device to some processor. Any data coming from that device is then routed to the appropriate processor. This avoids confusion about who gets interrupts.

V. The system from the teletype.

A traditionally neglected portion of system development is the executive, or top-level user program. All the glorious features in the world, such as a segmentation map, a sexy file system, and beautiful language processors do not appear to the user sitting at the teletype. Instead he sees (in the case of the DEC system) a group of strange commands which affect his process in sometimes flagrant and sometimes subtle ways (such as clobbering his core image when he asks for system status information).

The executive and command language should be as personalizable as possible. The user should be able to create macros or cliches which become a part of his environment whenever he is logged on. Ideally, the user should be able to easily EXTEND the executive capabilities in some uniform way. He should be able to add syntax and semantics for various peculiar operations he may do often (e.g. if he is providing a new service to users, such as a new debugging language or a statistics-taker, he wants to add the commands to the existing executive command repertoire).

Alsburg's ICL has been suggested as a possible starting point for a system command language.

It is perhaps appropriate to cite here some of the characteristics of the system which affect the user intimately. One of these is the ability to save and restore environments. This is particularly important if any amount of the user's work is imbedded in his active files. The BBN lisp system builds an active data structure which may represent many hours of work. A second important system feature is that errors of any variety cause minimum havoc. Errors should attack only the offensive user and not the sanity of the entire system. Except in very special cases, this should be easy to achieve.

The system should, for its own health, embody several other features. In some sort of privileged mode, a wizard should be able to gather extensive statistics about the running of the system, be able to tune the performance parameters of the system, and very possibly even be able to dynamically augment the system's capabilities. He might want to implement some new system function, and "patch" it into the environment on the spot.

It makes little difference whether we have a system which provides these functions directly, or one which has sufficient generality to enable us to implement the functions.

VI. The system guts.

The following list of system functions is not intended to be complete. It is intended to suggest those things which we think are important in the system, or which may meet particular requirements of the AU group.

1. DEC compatibility. It would be advantageous to be able to run programs generated by other PDP1Ø users. It might even be advantageous to be able to run DEC cusps!

2. I/O device service. Little can be said here, since each installation will have its own complement of devices. Secondary storage will be operated by the file system (see below). A user should ideally be able to debug new I/O devices and device service code as a semi-privileged or unprivileged user.

Terminal service is a source of heated and necessarily inconclusive debate. The solution set is as large as the user set. The user would appreciate several things:

--a minimum number of special characters glommed by the system to irrevocably mean special things.

--the ability to change the "character set" -- the map from incoming device bits to what is actually stuffed in the buffer.

--the ability to specify activation conditions rather carefully, such as a set of characters which require activation.

--the ability to have several kinds of echoing (immediate, deferred until program swallows input, and program driven) handled by the terminal service. It is also convenient to have the system echo special characters at the bidding of the user.

--TTY talk rings or crossbar switches for routing messages from teletype to teletype or teletype to program or program to several teletypes, etc. can be very convenient, particularly if a user who needs help or bug-fixing is on the other side of the continent.

--intra-line editing functions operating on the user's buffer if he requests that mode. Then the user needs know only one set of editing conventions.

--strange terminal devices should somehow get service. If we have displays which are not teletype surrogates, we should be able to service them as both graphic output devices and (perhaps) as the echoing device for keyboard input.

3. Processes. The mother's day committee agrees that processes are good for you. Their usefulness depends heavily on some other facilities:

a. User interrupts. These should represent exceptions ranging from device errors to system errors or warnings to user-generated exceptions to exceptions issued from other processes (with appropriate protection). Some of the specifics: a process may want to trap all system calls from its slaves (and thus simulate some other system, etc.), may want to know if the slaves' working sets expand, may want to bless any "stores" into a particular shared page (and thus monitor the writing into some data base), etc.

b. Interprocess communication. Is is this function that permits processes to be of any use whatever. Clearly one method of communication is to share address space. But this implies a degree of intimacy of processes which may not exist. For instance, we might want to create a process which is responsible for creating a special kind of listing. Users want to access this process by sending it text, or file names, etc. Processes should thus have a communication scheme which is identical to (or at least ressembles) file-system operations. Exchange of data among processes may imply activation or blocking of one or both processes.

4. File system.

a. Clearly the file system is page-oriented. The user may request a group of pages to be located in his address space -- that is how he gets some code to execute. But the efficiency of editors and assemblers may depend only on byte-at-a-time access to files. That access should ideally be handled directly by the system, and should be extremely fast. It would be helpful for editing if the file system were able to "bubble" files, allowing the editor to create new room for new text, and obviating a recopying of the file.

b. Subroutine files (a la 940) are of great use. If the I/O transfers and process communication are implemented in the same way, then subroutine files are almost automatic.

c. Protection and access privileges to pages of the file system should be consistent with all the access mechanisms in the system (e.g. access to processes). Users should be able to share pages of code and pages of data.

d. The file system should be maximally robust. If the system hard-halts there must be minimal damage to the file structures. Users feel file-garbaging more acutely than any amount of system inefficiencies (in fact, I am retyping this portion of text, due to a file-garbager).

e. If at all possible, all file operations should have a uniform format. Even if the file will physically be located on mag tape, ectape, or in the printer hopper, the data transfer operations should be the same. Difficulties arise in complete unformity, since neither dectape or mag tape are really page-oriented storage media, and may not have directories, etc. Special characteristics of the printer are obvious. The important thing is that all these I/O operations are in a strong sense file operations (as opposed to terminal operations).

5. Types of service and scheduling. The scheduling mechanism should not be intimately built into every nook and cranny of the system, since each user community will complain about the service enough to require at least extensive tweaking and probably extensive revision of almost any scheduler. Several features might help alleviate the possibility for complaint:

a. Allow the user to say as much as he can about the kind of service he needs, including estimating running time until completion of the current kind of computing, number of active pages required, etc.

b. Arrange a series of priorities, so that background compute jobs (if they really are just drones doing the monthly salary schedule, etc) can be distinguished from medium jobs (assemblers, compilers) and high- priority jobs (such as editors). Then a process of a particular priority can expect service of a certain quality. There will clearly have to be some system to insure that everyone's demands for service are within the computing capability of the system. A special case of these priorities is the "real-time" user, who desires a presumably small amount of compute time immediately upon unblocking.

c. Make the scheduler maximally tuneable to a particular set of job mixes, or to a particular swapping efficency, etc. The effectiveness of this will depend on the degree to which the performance of the rest of the system is independent of job mix effects. Ideally, fairly large changes in such numbers as average working set size, number of active users, number of active processes, number of opened pages, etc. should not encounter gross inefficencies in the system.

VII. Consider the alternatives ....

Now that we have loosely discussed a collection of requisites for the AU's system, we move on to discuss the various possibilities described in II above.

1. The DEC system falls far short of our description. It is not a page-oriented system, and the inefficiencies of running large jobs are felt acutely. The only other really major lack is the absense of a reasonable process structure and user interrupts. Carl Ellison at Utah is very seriously considering adding such features to the 10/50 system. He is willing to accept the overhead implicit in

the existing portions of the DEC system (command interpreter, swapper, etc).

Most AU's probably consider the effort required to really spruce up the DEC system to be great enough to warrant starting from scratch.

2. The MAC system is small, reasonably fully debugged, and in a sense, ready to go. The executive is probably insufficient for most users -- it is a doctored version of DDT which allows you to make system calls. The system is not at all DEC compatible, but conversion of reasonably large DEC cusps has been reported to be a "one night job," for an efficient hacker. They are in the process of implementing a TTY crossbar arrangement to route characters to the end of the earth. They have a good implementation of processes and user interrupts. The interprocess communication is just like I/O -- you give an IOWD pointer into the process's core. Several system functions are implemented with processes -- a statistics gatherer and general system-consistency checker. File operations do not differ markedly from the DEC system. The MAC system does not now enjoy paging, so any page-type files, or address space management are not available. MAC will soon include paging software in their system. The system is not prepared to swap, and insists that all executable code and windows on open files are kept in core. They estimate about 2 months wizard work would be required to install any swapping -- it would be a big job. There is very little protection in the system as a whole -- it is meant for a group of real friends. The scheduler is of course, designed for a non-swapping system. It records the

percentage of time granted to requested within the last 5 seconds, and uses this statistic to allocate CPU time.

3. The BBN system is almost fully coded, and is expected (by BBN) to be distributable by summer 1970. It implements most of the items discussed in VI. There is as yet no specification of their executive. In general, documentation on the proposed system is a little spotty. The exact set of system calls has not yet been specified.

Terminal service will be handled mostly by a system "process", scheduled like any other. Only interrupt-level echoing will be handled by the system directly. Hopefully, this will allow tailoring the teletype service to the user with somewhat less pain. The features such as TTY-TTY links, selective activation, special echoing, etc. are all being implemented.

The process functions have not been completely specified. They are expanding their user interrupt system to include some of the things mentioned in VI.3.a. The proposal for interprocess communication is a buffer with a "full" flag. The buffer can transfer data only one way. The status of the full flag is a possible user interrupt. There are grumblings that the system may implement repeated pokings of the buffer (block transfers), merely by iterating the single word transfer.

The file system is nearly complete. Subroutine files, in a slightly more general form than those of the 940, are planned. The file system will not be able to bubble files by itself. File names will be huge, and the file-name recognizer will require only as many characters of the name as are necessary to avoid ambiguity. The exact format of protection has not been disclosed. Use is made of some special features of the Bryant disc to help with error-checking the file system. The file system will remember date last written, date last accessed, and date last dumped. Temporary files will be implemented, they will disappear at logoff.

The scheduler is a fairly complicated device. It attempts to take account of memory usage, service request, past history, etc. They are particularly concerned with minimizing the number of activations of programs. It will be heavily tuneable, in part because BBN cannot acurately estimate what the important time-consumers will be in their system. They intend to provide real-time service.

4. The fourth alternative is to fashion the sketchy items in VI into a system design and to build it. This would require about 12 man years minimum, and would certainly require that all the participants in the project managed to get physically together.

The working committee has made the following judgements. Even extensive modifications of the DEC system would not yield a system as useful as either the MAC or the BBN system. Any effort here would be large in comparison to the gain.

It strikes us that we would need some pressing requirements beyond those provided by BBN or MAC in order to justify writing a system from scratch. Certainly there are techniques which we would enjoy seeing in the system implementation (e.g. Lampson's objects and capabilities), but their existence is not absolutely crucial to the usefulness of the system.

So we suggest that either the BBN or MAC system be adopted for distribution. The BBN system would present far fewer difficulties in preparing it for communal use: documentation of the system is proceeding with the writing of the system; minimum modifications to the real guts are needed (unlike the MAC system which will need the pager, the swapper, and probably some protection mechanism). There is evidence that the BBN system will submit to understanding and alterations more easily than the MAC system.

VIII. Procedure for acquiring a system.

If the BBN system is adopted, everyone will need a pager ressembling the BBN pager. BBN has of course offered to build them, or at least furnish prints of the design. Systems Concepts (Mike Levitt, Stew Nelson, and friends) have offered to build almost any paging box. They think they can do it for about $3Ø K. With reasonable technology (i.e. not DEC flip-chips) the pager should not be more expensive than that. It should probably be up to each installation which wants the system to come by the hardware necessary to run the system.

Distributing and maintaining the software will be a painful process no matter who does it or how it is done. Each installation will have to write some special code for their particular I/O gear (discs, teletypes, etc.), and this must be merged gracefully into the system that is common to all installations. Hopefully, the first distribution will not occur until such time that the installations will be able to write these I/O drivers without fear of having to rewrite them to conform to some subsequent release.

Changes, bug-fixes, and major updates must somehow be circulated to all users. There will always be the hackers who will seek to improve or enlarge the system, and there must be some mechanism of arbitrating such changes. The best approach seems to be to invest in one installation the responsibility for reviewing changes, checking out patches, and distributing them. They might even have some mercenary "trouble shooters" who knew the system cold, and in the lack of a wizard at the individual installation, could bail it out. This scheme sounds unfortunately like that attempted by DEC and (with a somewhat longer time-constant) IBM. It might be a real problem interesting some high-powered people in anything so pedestrian as maintaining a system. The one bribe than can be used is that the job will probably not be full-time, and the guy can have a good deal of time free to hack and do what he pleases.

IX. Comments on standards.

Even if all the AU's have identical systems, it may still be difficult to actively transmit useable programs from one installation to another. There will have to be some standardization of programming languages:

assemblers: MACRO, FAIL(Stanford).

debuggers: DDT,RAID(Stanford).

implementation languages: BLISS(Carnegie)

high-level languages: FORTRAN, SAIL(Stanford), LISP(BBN).

Perhaps a set of user service programs such as these should be agreed upon as reasonable programming environments, and make various installations responsible for maintaining a standard version for use by all the AU's. If the ARPA net really gets going, we could conceivably enjoy complete program compatibility for all but the most esoteric things.

It appears, however, that the combined problems of standards and of maintaining programs for a possibly diverse batch of users need some serious thought. The task of distributing this great new time-sharing system would be an excellent proving-ground for anyone's good ideas on the subject.

INTERFACE OF A PDP-10 TO THE COMPUTER NETWORK OF THE ADVANCED RESEARCH PROJECTS AGENCY

by

William A. Freeman

Submitted to the Department of Electrical Engineering on May 19, 1972 in partial fulfillment of the requirements for the Degree of Bachelor of Science

ABSTRACT

The computer network of the Advanced Research Projects Agency of the Defense Department is a nationwide interconnection of the computers at various projects funded by the agency. This allows these computers and members of these projects to exchange data and programs and run programs on other computers of the network. This work deals with the hardware necessary to connect a new computer to the network, specifically, a moderately heavily time-shared Pdp-10.

THESIS SUPERVISOR: Marvin L. Minsky

TITLE: Professor of Electrical Engineering

Table of Contents

INTRODUCTION

The network is maintained for ARPA (Advanced Reseach Projects Agency) by BBN (Bolt Braneck and Neueman) who originally designed the common hardware used and gave the specifications for connection to it. Full details of the hardware specifications are available from BBN to those groups who might connect. Many of the specifications are integrated into this work as necessary to explain design choices.

The network is organized as a collection of locations which are connected to the nearer of the other locations via 50 kilo-baud lines leased from the American Telephone and Telegraph Company. At each location is a computer of the mini-computer class called an IMP (interface message prossesser). In addition to being connected to the phone lines, each IMP has at least one teletype, and may be connected to a small number of computers located close to (within 2000 feet) the IMP. These nearby computers are called HOSTs and must supply the appropriate hardware to connect to the IMP on the IMP's terms. This hardware is called an IMP INTERFACE.

The Artificial Intelligence Laboratory (hereafter AI) at M.I.T. (The Massachusetts Institute of Technology) is funded in part by ARPA to carry on research involving computers. As such, AI has a time-sharing system called ITS (Incompatible Time-sharing System) running on a Digital Equipment Corporation

Pdp-10, which is a large general purpose computer. ARPA and AI have agreed that ITS should connect to the network. This work reports the design of the necessary IMP INTERFACE.

ACKNOWLEDGEMENT

I must acknowledge the guidance of Thomas Knight whose knowledge of the state of the art in TTL proved invaluable to me. Also Richard Greenblatt and Jeffrey Rubin were of great help in deciding how the device should appear to the programmer. I wish also to thank my thesis advisor, Professor Marvin Minsky, without whom all of this would have been in vain.

1. WHAT KIND OF LOGIC

There were really only two contenders for this position, Digital Equipment Corporation's Flip Chip logic (DEC Logic), and Transistor-Transistor-integrated-circuit-Logic (TTL). The only real point that Flip Chip has in its favor is that it is directly compatible with the Pdp-10 computer, by virtue of the fact that the Pdp-10 is made of Flip Chip logic.

TTL is much less expensive for the same functions than anything else capable of performing the job. This is because it has been very popular during the past several years, and is manufactured in huge quantities. It is faster than all but the exotic ECL, and is more than adequate in speed for the job at hand.

TTL is reliable. If its limits are respected, faliures are rare. Its noise margins are good. It plugs into dual in-line integrated circuit sockets, much less a source of trouble than DEC sockets. It is very small, so that the entire circuit may be compact, and worries of pick-up due to long wire runs are greatly reduced.

There is a wide variety of complex logic functions such as counters and shift-registers available in single dual in-line packs in TTL, obviating the need to build and debug such standard circuits. This also contributes to compactness in the finished device.

In the case of the AI Lab Pdp-10, even the difference in logic levels between the Pdp-10 and TTL is not a bar to TTL because we have a section of our I-O buss arranged to use TTL levels. This is because of the popularity of TTL within the lab for the construction of devices that will have to be connected to the computer. TTL compatible computer peripherals are also popular industry wide, so that even if AI did not have the TTL I-O buss, level converters to interface TTL to the machine are readily available.

2. TALKING TO THE IMP

This topic is covered in detail in BBN Report #1822, INTERFACE MESSAGE PROCESSOR, Specification for the interconnection of a Host and an IMP.

Specification for the two directions of communication is the same for each, i.e., each end of the interface contains a transmitter and a receiver capable of receiving the transmitter at the same end if connected to it, or any other transmitter built according to BBN's specifications. There are four signal paths between a receiver and a transmitter:

D for data, the data passes one bit at a time over this path.

RFNB for request for next bit, a flow control signal.

TYB for there's your bit, a flow control signal.

LB for last bit, to indicate the end of a message. D, TYB, and LB go toward the receiver, RFNB goes toward the transmitter.

The receiver when it is ready to receive a bit of data, i.e., when it has room in the interface and TYB is not true, raises (makes true) RFNB. The transmitter, when it both has a bit to send and sees RFNB come true presents that bit on D and raises TYB. The receiver, seeing TYB come true, copys the data from D and lowers (makes not true) RFNB. The transmitter, seeing RFNB drop (become not true), lowers TYB. This process is iterated to send data. In this way, if the transmitter is empty of bits and needs to be serviced by its computer or if the receiver is full of bits and needs to be serviced by its computer, the device in question may stop the data flow right at that bit, with no propagation delays to worry about and no re-synchronization problems such as would ensue with a synchronous system. There is also no trouble handling messages from machines of various word sizes as might arise with a byte oriented interface.

Data is handled in blocks of bits called messages which are of arbitrary length. A way of indicating the end of the message is needed. Sending a bit count isn't necessarily a good idea because the picking or dropping of a single bit would completely jumble all messages thereafter. When the Host sends the last bit of a message to the IMP it raises LB simultaneously with TYB. The IMP then appends a single one to the end of the message and enough zeroes to fill out the current IMP word (a small but variable number of zeroes) forming a packet (message plus the one and the zeroes). When the packet is sent to a Host the IMP raises LB simultaneously with TYB when it is sending the last bit of padding, indicating the end of the transmission. The Host interface then appends enough zeroes to fill out its current word. Then the software may simply look for the last one in the packet and thus find the end of the message.

When the Host is the transmitter and the IMP is the receiver the names of the signals are changed to:

HD for "Host Data"

RFNHB for "Ready For Next Host Bit"

TYHB for "There's Your Host Bit"

LHB for "Last Host Bit" Similarly, the signal names for transmission from IMP to Host are ID, RFNIB, TYIB, and LIB where the I is for "IMP". This allows us to talk about the specific directions of communication.

There is also a pair of lines in each direction on which a contact closure is provided by one end and looked at by the other. The contact closure indicates that the program to handle the network in the corresponding machine is running. If either of these lines should open, the message in progress during the open should be discarded. These are the IMR/IRT pair (IMP Master Ready/IMP Ready Test) to which the IMP's contacts connect, and the HMR/HRT pair (Host Master Ready/Host Ready Test) to which the Host's contacts are connected.

There are two kinds of line conventions for the actual transmission of signals. If the cable length between the Imp and Host is less than fifty feet, then a single shielded D. C. wire is used for each signal, with zero volts to ground being unasserted and five volts being asserted. In this case the Host is called "local".

For cable lengths up to two thousand feet the distant Host arrangement must be used. Each signal is sent over a balanced twisted 130 ohm pair using a one volt differential signal, centered around ground. The pairs are terminated at the transmitting end.

For distant Hosts, the IMP's end line drivers and receivers for the lines to the Host are completely floating to eliminate problems arising from the difference in ground potential. Transformers are used to accomplish this. The only effects as far as the Host interface is concerned are that the Host provides the ground reference, transitions of signals must take less than 100 nsecs., and the minimum time between transitions is one microsecond.

3. TALKING TO A PDP-10

i) Normal I-O

A Pdp-10 is a 36 bit wordsize machine and its input output structure is arranged to handle 36 bits in parallel, in either direction. The data is handled over 36 bi-directional, open-collector lines. These lines are also used to set and read flags called condition bits within each I-O device, which control various functions of the device and indicate the status of the device. Flow over these lines is controled by 20 uni-directional lines which go from the processor to the device.

14 of these lines are in complementry pairs and indicate which device is to use the data lines by presenting a seven bit device number special to that device. These lines are called I-O Select lines.

One of the remaining lines is called DATAI, and upon seeing this line asserted the selected device is expected to place data to be input to the computer on the data lines. Similarly, the selected device is expected to place its conditions bits for input on the data lines upon seeing the CONI signal.

Two more of these lines control transmission of data from the computer to the devices. DATAO CLEAR is given when the computer places data on the data lines. This signal goes away and DATAO SET is given. It and the data go away at the same time.

Similarly, the last two lines are used to cause the selected device to read commands and parameters special to that device from the data lines. These lines are called CONO SET and CONO CLEAR.

The Pdp-10 has a priority interrupt system, meaning that I-O devices can be ranked as to whether one of them can interrupt the I-O service routine of another. There are seven levels of priority, or Priority Interrupt (PI) channels. A device assigned to a PI channel may interrupt a routine running for a device assigned to a higher numbered channel, and its routine may be interrupted by a device assigned to a lower numbered channel. PI channels are numbered one through seven, non-I-O programs may be considered in some sense to be channel number eight. Interrupts happen by causing the execution of an instruction from a location different for each channel which may start a routine or may just be a single I-O instruction. This is accomplished in the hardware by using seven one-way lines in the I-O buss directed toward the processor, called PI lines. A device wishing to start an interrupt asserts the PI line for the channel to which it is assigned.

A device could be permanently assigned to a PI channel, but most devices have this specified by a three bit number sent to it via a CONO operation. The number of the channel assigned is the three bit number, zero being no assignment.

One recommendation on how to build the interface included the use of three PI channels, one for the data from the network, one for the data to the network, and one for error indications and last word in message indications. This was rejected because in such a heavily used time-sharing system as ITS, PI channel assignments are fairly well used up, with several devices on a channel already. Also, it is believed that the speed of network functions would be limited in the software in most cases, and the additional overhead involved in figuring out what the device wants is the smaller cost.

The one remaining signal in the I-O buss is I-O RESET, which is used after a power up or a crash to put I-O devices into a defined and docile state. For example, PI channel assignments are removed.

ii) Modifications to the AI LAB Pdp-10 affecting the design

One difference between the AI machine and other Pdp-10s is the existance of pseudo PI channels. These, essentially, are the division of normal PI channel number one into eight channels. One of these is the normal channel one, the others are used by doing what one would normally do to interrupt on channel one plus asserting the wire, which has been added to the I-O buss, associated with the particular Pseudo PI (PPI) channel on which it is desired to interrupt. A device on one PPI channel cannot interupt a routine running for a device on another PPI channel or on channel one, and a device on channel one cannot interrupt a routine running for a device on a PPI channel.

This was installed to reduce processing time for the faster devices normally assigned to channel one, which have been increasing in number of late. The speed-up comes from the fact that devices assigned to the same I-O channel must be polled in order to find out which device is interrupting when an interrupt occurs. This is obviously not necessary if there is only one device assigned to the PI channel, or PPI channel. Since it is frequently only necessary to perform one I-O instruction to service a device, the polling overhead can be quite an expense.

It is envisioned that, at least for the present, there will be only one I-O device assigned to a PPI channel. Also, for the moment, PPI assignments are to be wired in, even though it would be a simple matter to make these assignments in a similar manner to the way that PI assignments are made. A device that has a PPI Asignment (PPIA) can use it whenever it has a PI Assignment (PIA) of one.

It was decided that since the new source of interrupt channels had appeared and since the interface would be handling large volumes of data fairly fast during times when the system is heavily loaded, that there would be two PPI channels assigned to the interface, one for data coming from the network, and one for data going toward the network. For errors and end of message indications the interface would use normal channel one.

Another unusual fact about the AI lab I-O buss is that there are two processors connected to it. One is the Pdp-10, which runs the time-sharing system. The other is a Pdp-6, the predecessor to the Pdp-10, which formerly ran the system and is now retired to handling real-time jobs and other stand alone work.

In a system as experimental as ours, with users as casual as ours, it is necessary to be able to restrict the access to some I-O devices to one processor at a time. This selection is also necessary so that when a device wants to interrupt it only interrupts the processor it was working with.

To provide for this, two wires were added to the I-O buss, called PA and PB. These are always complementary and which one is asserted indicates which processor is using the buss at the moment. For normal I-O operations then, the appropriate wire is considered by a device as an eighth I-O select line. For interrupts, whenever the buss is not being used by either processor the lines are toggled at high frequency. A device only presents PI signals when the processor select line for the processor it is assigned to is asserted. A special piece of hardware remembers the signals and which processor gets them, and presents them to the appropriate processor.

Notice that a device might ignore the multiplexing if it does not use interrupts, or may be assigned permanently to one processor, or might be assigned by a toggle switch, or by a programmable feature in the hardware. All of these schemes are in fact used.

The network interface has a toggle switch to assign it to a processor.

4. CIRCUITRY, IMP TO PDP-10

Conversion of the serial bit stream into parallel 36 bit words is done in the obvious manner, i.e., by a 36 bit shift-register whose input is connected to ID. It is composed of five 8570's. The 8570 is an eight bit, serial in parallel out shift-registers. (see receive circuit, figure 1). The extra four bits are ignored.

The TYIB signal is connected to trigger a 74121, a one shot, monostable multivibrator, with its rising edge. The one shot produces a pulse of about 100 nsec. duration. This pulse shifts the shift-register. The pulse also clears a latch consisting of two nand gates, numbers 33-1 and 33-4. This latch is used as RFNIB, so this goes away as the data is taken. This latch also drives a 1-usec. delay circuit with its inverted output. The output of the delay circuit sets the latch again causing a new request for next bit.

An eight bit counter, formed of two 74193 four bit counters, is also driven with the pulse from the one shot, so that it counts bits received. After each 36 bit word is sent to the Pdp-10 this counter is loaded with the binary number "11011100" which is equivalent to negative 36 in the sense that after 36 counts it will reach all zeroes. When it reaches zero is the first time that the second order bit is zero since the counter was loaded. This is anded with the input to the delay circuit that sets the latch before this signal is given to the delay, so that when the 36th bit is received the latch will not be set again until the counter is reloaded, meaning that the Pdp-10 has copied the contents of the shift-register. This bit of the counter is also used as the flag to indicate that the receiver is "busy" (RBUSY). Its falling edge sets a flip-flop which is the indication that the receiver is "done" (RDONE). The one shot also clocks a flip-flop which will be set if the LIB line is asserted. This is the LIW flip-flop and indicates that this is the last Pdp-10 word in the message.

The LIW flip-flop enables the filling circuit. The filling circuit forces the input of the shift-register to be zero and connects the output of the latch normally used to generate RFNIB through another delay circuit to the input of the one shot. It also disconnects the latch from the RFNIB output, inhibiting RFNIB. This oscillates filling out the word with zeroes. Note that filling will stop when RBUSY goes away because this prevents the latch from setting.

It was decided to install a 32 bit mode, wherein the device would receive 32 bits and fill the word. This is useful because network messages are frequently structured as eight bit bytes, for control or for ASCII characters. This type of message is received in 36 bit mode. One out of nine bytes would be spread across two words, and while the Pdp-10 has instructions that make it easy to handle bytes, these only work if the byte is contained entirely in one word.

The middle four bits of the eight bit counter are anded together with a signal indicating that 32 bit mode is selected for the receiver (R32 Bit). All four of these counter bits are true between the 32nd and 36th count. The result of this and is used to enable the filling circuit, thereby causing the word to be filled out with zeroes after the first 32 bits.

To input a word from the device, the processor selects the device and presents DATAI. The appropriate inversions of the I-O select lines and the processor select line from the toggle switch are anded to form the I-O select signal (IOS) internal to the device (see figure 2). This is anded with the DATAI signal to form DATAI SEL which is used to gate the contents of the shift-register onto the I-O buss. It is also used to clear RDONE. Also, its falling edge triggers another one shot which clears LIW and loads the counter, thereby setting RBUSY.

5. PDP-10 TO IMP

Conversion of the 36 bit parallel Pdp-10 word to a serial bit stream was done using a 36 bit shift-register with its output being HD. It is made with five 8590, eight bit parallel in, serial in and out shift-registers. The extra four bits are ignored (see Transmission Circuit, figure 3).

After being anded with a few conditions, RFNHB is sent back out as TYHB. When TYHB falls it triggers a one shot which shifts the shift-register. The one shot also drives an eight bit counter composed of two 74193's. The counter is loaded when a word is sent from the processor to the device. It is loaded with a number such that the second highest orded bit becomes one after the 35th count if the transmitter is in 36 bit mode, or after the 31st count if the transmitter is in 32 bit mode. This bit of the counter is anded with LHW flip-flop (set for the last word in the Host message) and TBUSY (the "busy" flag in the transmitter) to form LHB. It is also anded with the pulse from the one shot and the result is used to clear TBUSY, which will occur at the next pulse after the counter bit is set. The falling edge of TBUSY sets TDONE (the transmitter "done" flag). The falling edge of TBUSY also clears LHW. TBUSY is one of the conditions anded with RFNHB to form TYHB so that when all of the bits in one word are sent TYHB is not generated.

IOS and DATAO CLEAR are anded forming DATAO SEL which is used to cause the shift-register to load itself from the I-O buss. It is also used to set TBUSY, clear TDONE, and load the counter. Its inversion is one of the conditions anded with RFNHB to form TYHB so that it is insured that the transmitter is really ready to send data, i.e., that the loading process is over, before it says so to the IMP by generating TYHB.

6. TRANSMISSION LINES

Since AI is a distant Host the differential scheme is used. The fact that the IMP end is floating greatly simplifies matters.

To receive the signals a pair of 8820 dual differential line receivers is used. They have enough sensitivity for the job, good common mode rejection reducing noise problems, and TTL outputs. In other words they're just right. They would be good for local Host operation too since the inverting input could be tied to a potential in the middle of the signal swing. They are convenient when the signal is really desired in the reverse polarity since one may just swap the inputs and save the inverter.

For transmission, two complementary TTL outputs are generated which drive nothing but the line through 68 ohm resistors (termination) save, perhaps, the inverter generating the other sense. This provides plenty of drive, the only thing not corresponding to ARPA specifications is that the signal is not ground centered. This, in fact, turns out to be a problem. The line receivers in the IMP suffer from a shortcoming common in cheap, discrete component differential amplifiers. If the voltage of the non-inverting input goes much above ground the output goes low regardless of where the inverting input is, due to a biasing problem. This problem was solved by raising the ground reference given to the IMP a few volts above the TTL ground. This does not bother the 8820 line receivers because they are capable of ignoring a common mode voltage in excess of 15 volts.

To drive a local Host transmission line one can use DTL or open-collector TTL with a pull-up. This is known to work because the MIT Dynamic Modeling group, a local Host on our IMP uses it.

The cable from the IMP plugs into the interface on a DEC card which contains the termination resistors and the voltage dividers for the ground reference. We have a dummy card which connects the transmitter and receiver together for debugging purposes.

7. ERROR CONTROL

Error really means catastrophy since the only error detected is that one machine or the other has forgotten that the connection between the two exists, usually indicative of a crash. The IMP has a timer that times how long it has been since the IMP program paid attention to the interface, and if it exceeds some nominal few seconds the timer opens the contact closure provided for the Host by the IMP. They recommend that the Host do the same, and we do.

The timing is accomplished by using a retriggerable mono-stable multivibrator with a very long time in the unstable side. Retriggerable means that the delay to return to the stable state may be restarted indefinitely without changing the output at any time. If the multivibrator reaches its stable state it sets a latch called the Host Error flag (HE). This flag being not set holds a relay closed. The relay is driven directly from the TTL and is, in fact, in a dual in-line pack itself. (see figure 4)

The mono-stable is retriggered by any CONI instruction to the device. CONI was chosen because it is the only I-O instruction to the device which can be performed without prior knowledge of the state of the device and not affect the state of the device other than the timer. This is useful so that the system can perform the instruction on a slow clock break pretty much unassociated with the network programs. The delay in the timer is about three seconds.

The IMP's contact closure is tested by grounding one side of the contact pair and connecting the other side of the pair to the input of an inverter with a pull-up resistor to VCC. The output of the inverter is called IMP Ready (IR). When IR goes low (contact open) it sets a latch called the IMP Error flag (IE). IR and the inversion of HE are anded together to provide the ALL READY signal which is anded with RFNIB and TYHB before they are sent out.

There is a flag called Don't Interrupt on Host Error (DIHE) whose inversion is anded with HE to generate an interrupt signal to give to the interrupt generation circuit (see section 8). There is another flag called Interrupt on IMP Ready (IIR) which is combined with IR and IE such as to choose which one of them may generate an interrupt signal. This is so that if the IMP is down one may arrange to be interrupted when it comes back up as opposed to having to check occasionally.

8. INTERRUPTS

The PIA is kept in a 3 bit register consisting of 3/4 of a 74175 (PIA0, PIA1, PIA2). There are four signals that cause interrupts, RDONE, TDONE, the signal from the IR/IE selection circuit, and the HE signal gated by the DIHE. These are ored together, anded with the processor selector signal, and inverted, providing a signal that says when not to interrupt. This is applied to the high order input of a one of ten decoder, the three bits of the PIA register go to the other inputs. This means that outputs zero through seven of the decoder will be asserted when an interrupt is desired, which of the eight controlled by the PIA. The outputs are inverting and open collector, and therefore suitable for driving the buss directly. Outputs one through seven are connected to PI lines one through seven respectively.

There are two signals that use pseudo PI channels, TDONE and RDONE. Since PPI channels exist on the Pdp-10 only, they are not to be gated with the normal processor select signal, but rather with a signal that indicates that the device is assigned to the Pdp-10. This is easily generated with another pole on the toggle switch used to select PA or PB as the processor select signal. PPI channels are also only used if the device is assigned to channel one. This is generated from the PI register and anded in the same gate that generates it with the "assigned to Pdp-10" signal. This signal is anded individually with RDONE and TDONE to form the signals to drive the PPI lines. RDONE is additionally anded with the inversion of LIW so that the receiver interrupts on normal channel one for the last word in a message.

9. CONDITIONS REGISTERS

CONI is anded with IOS forming CONI Sel, which is used to gate various signals and flags within the device onto the IO buss (see figure 5). Figure 6a shows how the bits are arranged in the word that comes into the processor (only the low order 18 bits are relevant).

CONI SEL is also used to retrigger the Host error delay multivibrator (figure 4).

CONO CLEAR is anded with IOS to form CONO Sel. This is used to cause condition information to be taken from the IO buss. Some flags are set by one bit on the IO buss being one and cleared by another, or just one of those (RDONE, TDONE, R32 bit, HE, IE, LHW). Others are set to the value of a bit on the IO bus, i.e., loaded from an IO bit (PIA0, PIA1, PIA2, PIHE, Interrupt on IMP Ready). (see figures 1, 3, 4, & 5). Figure 6b shows which bits do what in the word coming from the processor.

10. LEVEL CONVERTERS

The level converters are ones made by System Concepts of San Francisco. They are in one-way types for either direction and in two-way types. The two senses of inversion are independently selectable for each direction of each card of four converters.

Two-way converters are used on the I-O buss data lines and one-way converters are used for all else. Signals headed toward the processor are arranged to be ground asserted and signals headed to the device are arranged to be ground not asserted. The two-way level converters need a signal to tell them which way they are going which is generated in the device by oring CONI and DATAI.

The decision to use separate level converters as opposed to the TTL section of the I-O buss was made in the higher levels of the AI lab design staff, giving as reason that devices on the TTL I-O buss must be wired in while a device as important as the network interface should be plugged in as on the DEC I-O buss. This was convenient as the ability to select inversion, and the fact that one need not worry about loading of the buss eliminated the need for 20 inverters in the device itself.

11. PHYSICAL DESCRIPTION

The interface itself fits onto one Augat wire-wrap, 60-16 pin dual in line socket board. This is mounted along with a row of DEC sockets for plugging in the I-O buss, the level converters, the cable to the IMP, and a set of lamp drivers used to display conditions information and which processor is selected. The whole thing and its power supplies and lamps are mounted in a Pdp-6 vintage DEC cabinet.

12. REPORT OF OPERATION

There were a few wiring errors but for the most part the device ran as designed. The only change was from using DATAO SET to using DATAO CLEAR to read information from the buss. A timing marginality existed in that DATAO SET goes away "simultaneously" with the data on the data lines, causing occasional errors. The device is currently operating, software is still being debugged.

13. GENERALITY OF THE DESIGN

The basic interface is quite general. By changing the shift-register length and the numbers loaded into the counters, the word size of any machine may be matched. Level conversion is simple and, in fact, TTL levels are used in some machines. Inverting the sense of the bits in the data word is simple, swapping the normal and inverting outputs of the shift-register for the transmitter and at most the addition of one gate in the receiver. The "Busy/Done" organization is useful with most computers. Only the interrupt system and means of handling conditions bits need be changed in the basic design for most other computers.

The device could be used pretty much as is on most other Pdp-10s, and, in fact, there is another copy being built at the time of this writing for the MIT Mathlab Pdp-10. The circuitry for the pseudo-priority interrupt system could be ignored or straightfowardly changed to additional PI assignments by the use of a second device number. If another group wished to build one they could probably obtain the prints and wirelist for the Augat board from AI.

Patrick Greussay

Tools: A Systems Programming Environment

Nathaniel Mishkin
Steven Wood
John R. Ellis

Department of Computer Science
Yale University
New Haven, Connecticut 06520

This paper presents the design and implementation of a computing and programming environment that is different from environments found in industrial research centers, academic systems research, and university computer centers. The environment represents a commitment to using state-of-the-art techniques in the real world. It is not a concept, not a prototype, not (in the traditional view) a piece of research. It has none of the perfection of unimplemented academic visions. It is grubby and gritty and solid and real. It is, if you will, computer engineering rather than computer science.

We call this environment "Tools."

We are going to talk a lot, in detail, about one particular operating system, DEC's TOPS-20. We believe that any discussion of a computer environment or software engineering must necessarily include the details of the software being engineered. In building a real environment it is not possible to magic away all the unpleasantness of real software. Many of our design decisions were based, not on grand, pleasing algorithms and techniques, but rather on compromise. Engineering, building real structures in the real world, always requires compromise. The object is to show that one can compromise without collapsing. We didn't achieve any miracles. But we did succeed in building a computer environment that provides state-of-the-art facilities on heavily loaded machines to a large community of users who are not systems programmers. The general wisdom has been that that's impossible, or too hard, or not worth doing, or not original research, or that it will be irrelevant in a short time because everyone will have vast amounts of personal computing power, or that it's somebody else's job, or that nobody but systems hackers will use it anyway. We did it. Hundreds of people use it every day. It exists.

1 Raw Materials: TOPS-20

Right at the beginning of our attempt to bring the state of the art in computing environments to a group of real, ordinary users, the very first point on which we had to compromise was holism. The conventional view is that serious systems design must be done from the ground up; if you want to do it right, you start from zero.

No such luxury was available to us. The authors of this paper and all the others who worked on the Tools project were graduate students. Our time and our resources were limited; nobody offered us the chance to undertake total design. So we took as our text the wry words of Ecclesiastes: A living dog is better than a dead lion. Our living dog was TOPS-20.

TOPS-20 is a multiprogramming timesharing system for DEC-20s. The basic structure is a collection of "jobs", each job being itself a collection of hierarchically organized processes (or "forks"). Each user who logs in creates a job. At the root of his process tree, running as the top process in his job, is the command handler, the EXEC.

The TOPS-20 EXEC has two ways of handling commands. Either it executes the command itself or it creates an inferior process to execute the command. The reason for the difference is that creating, loading, and starting a process is slow work. The authors of TOPS-20 felt that common commands like COPY, RENAME, and DAYTIME should respond quickly. The only way to make them fast is to have the EXEC execute them directly. (Such commands are called, logically enough, "EXEC commands".) For any other command, the EXEC creates a process, loads the program into it, starts the process, and then waits for it to halt. After the process halts, the EXEC resumes accepting commands. If a subsequent command requires running another program, the EXEC destroys the the existing process and creates, loads, and starts up a new process.

Probably the most distinctive feature of TOPS-20 is its powerful command parser. From the user's point of view, what the command parser gives him is the ability to abbreviate and -- much more important -- the ability to get instant information with only a keystroke or two about what command options are available to him, what arguments are expected of him, and what defaults have been set for him. These results of the command parser are collectively called, in TOPS-20 jargon, "recognition"; together they make TOPS-20 the interactive operating system *par excellence*.

So much for the virtues of TOPS-20; had it been the perfect operating system we would have had nothing to do.

The computing environment that TOPS-20 creates has two fundamental problems: Users have no way to get at many important features of the operating system. And there is no uniformity whatsoever among the TOPS-20 utility programs. Together these problems serve to cripple the user. Much of what he wants to do he cannot because TOPS-20 won't let him; much of what TOPS-20 will let him do he cannot because TOPS-20 makes it too hard.

TOPS-20 includes such features as I/O redirection, argument passing, and the suspension and execution of multiple processes. But the EXEC has no provision for using these features. For instance, TOPS-20 has the concept of primary input and output streams, which default to the terminal. Operating-system calls can redefine these streams to refer to files -- but the EXEC cannot.

Or again, the EXEC can do argument passing and use the command parser only for EXEC commands. If a program that runs in an inferior process wants to use recognition, it must use separate lines for subcommands and arguments and interface to the command parser via a system call.

Or again, as a multiprogramming system TOPS-20 allows for the hierarchical organization of multiple processes, each running independently in its own address space, but there is no access to this facility from the EXEC. The user must constantly create new proceses and re-establish the state of every process every time he wants to resume it, redefining all the functions defined in an interpreter, rereading into memory all the files read in for a text editor.

The reason for these weaknesses in the EXEC is simple. Many DEC utilities were originally written for use on TOPS-10, an operating system for the PDP-10. TOPS-10 programs run on TOPS-20 in "compatibility mode", simulating TOPS-10 system calls. Compatibility mode permits no I/O redirection, no argument passing, no multiple processes -- and many of the standard TOPS-20 utilities, including the linker and the assembler, run in compatibility mode.

In judging TOPS-20 as a computing environment, the lack of uniformity in utility programs is a more serious problem even than the artificial hobbling of the EXEC. Users learn to live without what they absolutely cannot do. But they do not learn to live with inconsistency. The lawless jumble of TOPS-20 utility programs with their wildly varying and disruptive command syntax prevents users from grasping the underlying regularities of the system. They are so frustrated at the surface level -- they are made to feel so stupid and incompetent -- that they lose all taste for computing. The system is their enemy.

The distinction between EXEC commands and programs that run in inferior processes is wholly lost on most users. All they know is that some things work sometimes and don't work other times. And the interface between the command parser and programs that run in inferior processes is the cause of endless confusion among users. The program need not be concerned with how to parse, say, a date, or a file name, or a list of keywords; it simply provides the command parser with a list of options for every field in the subcommand. The syntax for subcommands and arguments can therefore vary wildly from one program to the next. Nothing the user learns about the functioning of one program is any use to him in mastering another.

And again, to look at yet another instance of the problems caused by compatibility mode, the file-name syntax for TOPS-20 is much broader and more flexible than for TOPS-10, but compatibility requires file names with TOPS-10 syntax. To the user whose valid TOPS-20 file-name syntax is suddenly rejected by the simulated TOPS-10, this situation looks like anarchy. He feels upset and angry and discouraged.

On the whole, TOPS-20 made for a fairly typical example of a real-world computing environment. It had some good things about it. Many of the bad things served as cautionary tales for us; since we had no choice but to adopt an incremental approach, it was bracing to have the hateful example of compatibility mode ever before our eyes. Above all, it was there.

2 Raw Materials Part II: The User Community

TOPS-20 was half of what we inherited at the beginning of the Tools project; the other half was a community of users.

There are two DEC-20s in the Yale Department of Computer Science. One of the machines typically has fifty users logged in at any one time, the other thirty. Departmental users include faculty members and graduate students in artificial intelligence, cognitive psychology, and numerical analysis. Many of the department's undergraduate courses are taught on one of the 20s, and undergraduate majors use it for independent research. The departmental staff use the machines for wordprocessing and for clerical and administrative work. The School of Organization and Management shares one of the machines, using it for research, education, and administration. And the department actually sells wordprocessing and computing services to other departments throughout the University. During the school year there are over 1400 users on the two machines. Few of them are systems programmers; most have never had any other experience with computers. Some are physicists, some are historians, some are philosophers, some are secretaries and clerk-typists.

Nor are there ceilings imposed on any users; fast display terminals, for instance, are universal, and wordprocessing users have access to the full range of computing facilities, not just to an artificial subset.

Obviously another point on which we had to compromise in designing a computing environment was isolation. Traditionally it is almost an article of faith that the true systems programmer must work by himself, in splendid solutide, protected from the hurly-burly of ordinary users.

However pleasant and gratifying such isolation might have been, it was not available to us. We were not offered the option of throwing 1400 users off the system and making it into a private playground for ourselves. If we wanted to work at all, we had to do it in the midst of a busy group of users with their own habits and their own needs and their own priorities.

Many believe that any work at all on timeshared systems is now moribund. Soon, they assert, immense amounts of computing power will be freely available to every individual in the form of the so-called personal computer. On economic grounds alone, we doubt it. The design success and the extremely low cost of DEC's VAX family suggest that medium-sized timeshared computers will be competitive for a number of years. If so, timeshared computing offers a fertile field for research -- especially to the researcher who is willing to consider the sociology of mixed groups of users a problem in software engineering. It is not at all clear how to bring current computing techniques to large systems with heavy load averages and users at many different levels of expertise, with many different, sometimes conflicting goals.

In dealing with users as in dealing with the operating system, we settled on a philosophy of compromise. We decided to try to make the best environment we could realistically build available to the widest community of users we could realistically serve.

In practice that turned out to mean that Tools programs are designed for users who range from novice to expert. By novice users we mean people who are in the process of learning the craft of using computers. We expect them to use the system regularly, but for tasks other than systems programming. We have to admit that Tools programs are not suited to truly naive, dead-end users -- people who use the machine infrequently, only to get some particular task done, and who avowedly do not want to learn about computers. No system, we decided, can at the same time hold the hands of users who do not want to learn and yet be terse and efficient for experts. Even designing an environment for expert users to share with eager, intelligent novices is a difficult task. It requires the expert user who is designing the software to put himself in the position of the novice. This is hard because the expert has developed a cognitive model of what's going on when he interacts with the computer. The novice user, by definition, does not yet have such a model, or has a model that is incomplete -- or wrong.

We left as an open problem the design of systems for users who want to treat the computer as they would an electric razor or a toaster oven: Either it works so I don't have to think about it or I want nothing to do with it.

3 The Three Basic Tools

Our goal was to build a state-of-the-art computing environment for ordinary mortals to use. Our raw materials were TOPS-20, running on two heavily loaded DEC-20s, and a group of users with other things than systems programming on their minds. Our time and energy were severely limited. So what did we decide to do?

The three major components of the Tools environment are a process manager called MUF, a full-screen editor called Z, and a session logger called SM [Ellis81, Wood81].

MUF (for Multiple User Forks) takes the place of the EXEC at the root of the user's process tree. Unlike the TOPS-20 EXEC, MUF can execute and suspend multiple processes. MUF allows the user to maintain as many different contexts as he likes. Typically MUF runs one process interactively at any one time and either suspends all the other processes or runs them asynchronously, taking their input from and redirecting their output to files.

Under MUF each inferior process is associated with a single character -- in the following picture, for instance, the editor with A, the EXEC with E, and a documentation process with T.

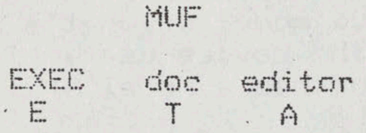


Figure 1: A Very Simple MUF Running 3 Processes

Z, the screen editor, is the linchpin of the whole Tools environment. (Z stands for nothing, by the way; the very first Yale screen editor was called E for Editor and began a series of editors with single-letter names, of which we expect Z to be the last.) It is very powerful; the terse, staccato listing of all its command options without explanation or instruction takes nearly two thousand lines. Nonetheless it is so natural and comfortable to use that rank beginners, people who have never previously logged on to a computer, find themselves able to do some productive work with it after only an hour or two of on-line tutoring. Three things make Z easy to use: the file model that it presents to the user, its uniform command syntax, and its interactiveness.

Z presents a file as if it were a very wide, very long scroll of rubberized paper. The user does not see a file as a stream of characters that include newline and tab; he sees the file as a very long, elastic array of very wide, elastic lines. There are commands for positioning the display window anywhere on this scroll and for positioning the cursor anywhere within the display window. The cursor is the focus of each editing operation. If the user positions the cursor at existing text and types, he overwrites the contents of the file; if he positions the cursor at what he perceives as blank space and types, the editor automatically extends the record or the file to accommodate the extra characters. What the user sees on the screen at any moment is precisely what is present in the corresponding section of the file.

Z's commands are assigned to control characters; except for the cursor keys, little use is made of special function keys since they require the user to move his hand away from the typewriter keyboard. The object has been to make using the editor an extension of touch-typing on a regular typewriter. The commands take a rich set of arguments in a variety of formats. The <arg> command, a single keystroke, initiates argument mode, which then terminates with the editor command that takes the argument. Every command also has a default action when invoked with no argument and many have second defaults invoked by typing the <arg> keystroke and the command keystroke with no argument in between.

In addition to text and numerical arguments, Z recognizes two kinds of cursor arguments, the box and the stream. A box argument functions like a scissors, snipping out a rectangle of text for moving, deleting, or manipulating; a stream argument functions like a reader, moving from left to right till it hits end-of-line, starting over again at the far left, and reading right again. Box arguments make it easy to handle text in blocks and columns, stream arguments make it easy to handle text in phrases and sentences.

Z tries hard to keep the time between any keystroke, no matter how complicated a command it invokes, and visual response to the keystroke at less than a second. For graphic characters the visual response is just the character itself, displayed at the current cursor position, and Z responds to graphic characters and hardware cursor keys very quickly by letting the TOPS-20 monitor [DEC80MCRG] echo them directly, without causing an expensive process wakeup. Some operations, like reading and writing a file or searching for a string, can take a long time on large files. To keep the user from wondering what the editor is doing, Z maintains a line counter at the bottom of the display that keeps track of the progress of the editor command as a small, friendly gesture.

Although MUF maintains the states of several processes, it does not maintain the display output of those processes. This lack led to the development of SM (for Session Manager), a one-window version of the INTERLISP Programmer's Assistant [Teitelman77]. There is only one window because our terminals have small screens and because TOPS-20 makes such windows expensive. Any line-oriented program can be placed in the window.

Figure 2: LISP Running under SM

SM has two modes: In edit mode, the user can give any normal Z editing command, such as scrolling or looking at other files. In SM mode, every keystroke the user types is input to the program running under the editor, and any program output is simultaneously displayed on the screen and recorded in a transcript file.

The user can switch modes with single-keystroke commands. In SM mode, a program running under the editor behaves exactly as it does when run under the EXEC. Whenever the user types the up-cursor key -- in the middle of executing a program -- SM suspends the program and switches back to edit mode with the transcript as the current file. Visual fidelity is always maintained between the display and the transcript file, so that switching from SM mode to edit mode is instantaneous, with no need to rewrite the screen.

The SM transcript can be edited like any other file. The user can extract text from the transcript and give it as input to the program without retyping. SM remembers the location of the most recent input lines read by the program, so it is very easy to move up into the transcript file, make a correction to an input line, and then give the corrected line back to the program. Users of interpreted languages are particularly fond of SM because it provides a history of interactions with the interpreter -- a history that the user can correct and refine in small successive steps.

SM works well with any program that does line-at-a-time I/O. It is particularly useful for browsing through groups of files. For example, a user can run the Tools LS program to list a group of files in a compact multi-column format, which SM records in its transcript. The user can then move into the transcript file and use a two-keystroke editor command to view each of the files in turn. At the same time he can use another two-stroke command to submit any of the file names as input text to a DELETE command that he is building up under SM. In a while, when he has reviewed all his files, he can use the command line he built up to delete the ones he no longer wants. At no time does he ever type a file name; at no time does he use a command that he cannot use with any other program under SM.

4 A Tools Demonstration

It is difficult to demonstrate a dynamic, interactive environment on paper but, given the technical problems associated with supplying a video terminal and a DEC-20 as an appendix to every copy of this paper, we will try to simulate an on-line demonstration of the Tools environment.

Picture a user who is writing a BLISS program in the Z editor process under MUF. Another process under MUF has an EXEC. Yet another has the on-line documentation for the BLISS run-time library. (We don't believe in hard-copy manuals; every Tools program is documented on-line and every document interactively indexed on-line.)

```
       MUF

 EXEC  doc  editor
   E    T      A
```

Figure 3: A Simple MUF

Now suppose that the user wants to look at the documentation on the BLISS PRINTF routine. Three keystrokes take him to the documentation fork:

```
^X T ^Z
```

(We represent the control shift with ^.) ^X is the <arg> command; ^Z, the <exit> command, passes the letter T to MUF; MUF invokes the T process. Two keystrokes plus the name of the routine display the documentation:

```
^X printf ^P
```

^X is <arg>; ^P positions the display window at the documentation for PRINTF. The single keystroke ^Z, <exit>, then suspends the T process and resumes the editor process right where it left off.

Now suppose the user has a slightly more complicated MUF structure:

```
                MUF

     EXEC  doc  editor  LISP
      E     T      A      L
```

Figure 4: MUF with 4 Forks

This time he is in the L process, running the LISP interpreter, and he has just discovered a bug in one of his functions. Typing the single keystroke ^A will switch him to the editor process, which will display the last file he edited, the file containing the function. (If another file were last, two or three more keystrokes would take him to the correct file.) He makes the correction, marks off the corrected text by moving the cursor over it, and leaves the editor. The editor tells MUF to resume LISP and supplies as input the text marked off with the cursor. The function has now been corrected with very few keystrokes and very little effort, both the LISP and the editor process have been maintained intact, and the user can proceed with debugging.

While MUF allows a user to have multiple forks executing simultaneously, it is often more convenient to use Z's background-compilation facility. This facility allows the user to start up a compiler process behind the editor process. The user can continue editing while the compiler is running. The editor then informs the user when the compilation finishes and displays any error mesages one at a time, positioning the cursor at the location of the error in the source file as it displays the message.

This facility works with any compiler, including document compilers like Scribe, and has been extended to support other asynchronous activities. For example, the user can run a program behind the editor that searches for a text pattern through a set of files; when the search is finished, the editor will step him through all the files that contained a match, positioning the cursor at the location of each match in turn. Or he can run the network file-transfer program behind the editor or use a background mailer to mail a file he has just edited without ever leaving the editor.

Now let's give our user a really sophisticated MUF!

```
          TOPMUF

EXEC  doc  editor  telnet
 E     T     A       F
             SM

            MUF

        EXEC  LISP  mail
         E     L     H
```

Figure 5: A Many-Splendored MUF

This time he is running LISP under a MUF under SM and encounters a problem that he cannot so easily correct. With a single keystroke, ^H, he switches to his mail fork and sends a message to an older and wiser friend, asking for help. Resolving to put the problem out of his mind for a while (and secure in the knowledge that his LISP fork is preserved intact), he goes up to the editor, again with a single stroke, and begins work on another file. A portion of his latest research report is ready so with two strokes he sets a document compiler to work on it behind the editor. He then decides that he needs to look at some information he has stored on another machine. Three keystrokes take him up to his telnet fork and log him on to the remote machine by means of an automatic initiation file that gets read whenever the fork starts up. A minute later, having read the information he wanted, he is back in the editor. A system message tells him he has mail and two keystrokes take him down to the mail fork -- and so on and so on.

The sensation of working in this environment is like the sensation of working at a well-organized desk or cooking in a well-organized kitchen. MUF, Z, and SM all dovetail, giving the user the sense of seamless, effortless transfer of control.

5 The Tools Programs

The Tools environment is dynamic. Users report bugs and frequently make suggestions for new features and programs. The Tools programmers (all graduate students) fix bugs quickly and accept suggestions as time permits. Changes can come quickly and potential users of Tools programs are warned that they must be able to keep up. Every single program is documented on-line in a uniform format, documentation is kept up to date, and announcements of changes are made via a "bulletin board" facility that users examine automatically whenever they log in. Programs such as the Z editor automatically display change listings; every time a user runs Z he gets a list of any changes made since the last time he ran it.

Our response to the demand for stability is that a stable environment soon becomes ossified. The problems caused by an environment that changes are far outweighed by the advantages of an environment that responds.

I/O redirection is possible with all Tools programs. With a simple entry on the command line, input to a program that is normally read from the terminal can be read from a file. Another simple entry redirects terminal output to files.

Virtually all the Tools programs accept arguments on the EXEC command line. Users view programs that run in inferior processes as being just like built-in EXEC commands -- as atomic transactions. The user does not have to think about whether he is in the EXEC or in some inferior process.

Tools programs embody the principle that common appplications should be simple to express. The programs have been designed so that their default behavior, the result of supplying no arguments to the program, is what we believe most users want most of the time. For example, a program that displays disk-space usage will assume, if no argument is specified, that the user wants to know about his own.

There are over a hundred Tools programs (see the appendix for a summary). They fall into two groups, job controllers and file manipulators.

Job controllers are programs used during a session to manipulate various aspects of a user's environment, often by manipulating processes that contain other programs. The primary job-control tool is of course MUF. Other job controllers are the local version of the EXEC, various EXEC-like tools, and DO and SH, our command-file processers.

The Tools EXEC is not vastly different from the EXEC that came with TOPS-20. We had no stomach for making large-scale changes to 35,000 lines of assembly language. Instead, myriad minor changes made the EXEC easier to use and integrated it into the Tools environment. For example, we added more convenient abbreviations, made the syntax of many of the commands more consistent, provided good defaults for command arguments, and implemented a few convenience commands. A simple but important change was to allow recognition parsing of file names and user names on the command lines of arbitrary programs. All of these changes are individually trivial, but together they make the Tools EXEC much friendlier than the original. We made our major changes elsewhere by implementing new programs such as MUF, Z, and SM; other installations attacked the EXEC directly, and we discuss below the multi-forking EXEC and the PCL EXEC. Other installations approached the EXEC with the "big hack"; we decided to pay attention to the details.

INEED is a good example of an EXEC-like job-controlling tool. TOPS-20 has a tree-structure file system, with each directory in the tree assigned a disk quota. There is no way to reassign quota directly from one sub-directory to another; you have to go through the root. Users have to shuffle space up and down in their directories and subdirectories, painfully. INEED simulates a better system. If a user wants more space in one of his sub-directories he types "INEED <sub-directory name> <incremental increase>". INEED automatically walks over the directory tree, moving unused quota up from subdirectories to the root directory and then down to the specified directory.

MAKDIR is another program that makes up for an EXEC deficiency. The EXEC's command for creating directories is verbose and requires multiple command lines. Most users do not know what many of the command options mean. Typically, a user wants to create a sub-directory whose parameters are the same as his other sub-directories. MAKDIR runs in two modes: The first saves up the characteristics of an existing directory, the second creates a directory with the saved characteristics. Once a user has established a characteristics file by running MAKDIR in the first mode, he can create a new sub-directory whenever he likes simply by typing "MAKDIR <directory name>", which runs the second mode.

DO, to look at another job-controlling program, allows a user to execute commands from a file. The EXEC provides a similar facility, but it is not very powerful. DO allows the command file to be parameterized by command-line arguments; the EXEC allows no command-file parameters. Whereas an EXEC command file can contain only EXEC commands, a DO command file can contain input to be read by programs as well as EXEC commands. SH is a slower but even more powerful version of DO; it can redirect its primary output to files, it does not echo commands as it executes, and it can recursively invoke other SH command files.

File manipulation is perhaps the most traditional task for computer utilities. Such utilities are used to create, edit, search, and compare files. There are over twenty such programs among the Tools utilities -- Z, of course, most prominent among them.

DIF, the Tools program that compares two files, is a good example of the difference in approach between a vendor-supplied utility and a Tools program. DEC's file comparer is called FILCOM. To compare two files, file1 and file2, you type "filcom". The FILCOM program then begins and prompts for arguments. You must specify the input files and output to terminal, and you must use a format that, while not unique to FILCOM, is slightly different from the format for other file manipulators: "tty:=file1,file2". Then once you get the difference listing you must type ^C to get FILCOM to terminate. To do the identical task with DIF requires the single command "dif file1 file2". FILCOM has problems with some files because it runs in compatibility mode, while DIF can take any valid TOPS-20 file name. FILCOM produces a listing that is less informative than DIF's listing.

TRANS, another file manipulator, is a powerful tool for transforming one file into another. It takes two arguments: a regular-expression pattern, which describes logical "records" in an input file, and an output format, which can contain text and references to parts of the input record matched by the pattern. The regular-expression language is a standard among the Tools programs and is used by many other programs; it is comparable to but far more powerful than the regular-expression language on UNIX. TRANS's input and output default to the terminal but can be redirected with the same I/O redirection syntax as all other Tools programs. A typical application of TRANS would be to transform a text file consisting of information about people, including, say, name, address, phone number, and birth date in some standard form, into a file of nothing but names and addresses to be printed on mailing labels for a mass mailing.

6 Command Syntax for Tools Programs

The issue of command syntax makes a good example of the kind of compromises we chose and the reasoning behind them.

Virtually all Tools programs use the same very simple command syntax. The similarity of the syntax makes it easy for people to learn how to use new programs. The user who feels confident about his mastery of a piece of syntax like

```
        usage:  LS [filespec]
```

can see something familiar and reassuring in

```
        usage:  LS [-codtbrwsupi] [filespec]
```

and eventually work his way up to

```
        usage: TRANS [-cdglmt]<format description><output
description><filespec>
```

Uniform syntax encourages users to expand their understanding and their skills.

This syntactic style was adopted from Unix. From the user's viewpoint, its main virtues are simplicity and uniformity. From the implementor's viewpoint, the syntax is trivial to implement reliably. He can devote his time to making a program work, instead of making it parse complicated syntax. The simple syntax also makes it easy for one program to invoke another without needing a complicated parser as intermediary. These are not insignificant considerations for environments that are perennially short of systems programmers, and where the choice is often between a program with simple syntax or no program at all.

The significant drawback of the Tools syntax is that it doesn't take full advantage of the TOPS-20 command parser. Typing a "?" after the name of a Tools program will not tell the user anything about the expected syntax for the program, as it will for an EXEC command. Likewise, recognition of abbreviated arguments will not work. Invoking Tools programs with the wrong argument syntax or with no arguments will print a reminder such as:

```
usage:  LS [-codtbrwsupi] [filespec]
```

but this reminder won't help the user remember what the non-mnemonic single-letter switches mean. Nor is the reminder very interactive; the user can't ask for syntax help in the middle of a typing a command line. For Tools programs, the EXEC does fill out abbreviated file names and user names on the program command line; this is handy because most arguments to Tools programs are file names, but it doesn't help the user who's completely in the dark as to what the swiches mean and what arguments he should be giving.

A fair amount of good experimenting has been done on interactive, extensible command parsers. Lantz [Lantz80] describes an ambitious parser that was part of a distributed computing system; unfortunately it never progressed beyond the experimental stage. MXEC [Ash81] and the PCL EXEC [PCLEXEC] both try to provide extensible parsing; for practical reasons, both are unsatisfactory (they are discussed in detail later).

Obviously the best of all possible worlds would have been simple, perfectly uniform syntax combined with full access to the command parser. We didn't have the time or the resources for the best of all possible worlds, but neither did we throw up our hands in despair and say that since perfection was impossible we wouldn't even try. Simplicity and uniformity were clearly within our reach; all they required was disipline.

7 A Tools-style Network

A good example of the Tools approach to software engineering is how we implemented our local network.

The Computer Science Department at Yale connects all its computers -- two 20's, two VAXes, and two PDP-11's -- via Chaosnet, a high-speed, Ethernet-like local area network [Moon79, Metcalfe76]. (A number of Apollos will be added shortly.) The initial hardware and software for the network came from MIT and were modified here. Once the basic network was up and running, along with the user-level software for electronic mail and file transfer, we started to think about what we really wanted to do with the network.

One of the first observations we made was that most existing networks, such as ARPAnet, CHAOSnet, and Ethernet, are painful to use because they make the network a completely separate world. In order to send mail to someone, you have to know the name of his computer account on the host that you wish to send the mail to -- which may not be where he is currently working. Transferring files requires you to run a special file-transfer program, log in to the remote machine, and then move the files one at a time. Beyond transferring mail and files, these networks are seldom used because they are so poorly integrated with the rest of the user environment.

Our first step, then, was to develop the mechanisms that would enable us to integrate our local network into the Tools environment. We needed some sort of glue to hold our heterogeneous collection of machines together. So we designed and implemented a centralized network data base filled with information of network-wide interest. The data base contains information about people (such as user IDs, phone numbers, and office addresses); it contains information about terminal locations and network host names and addresses; and it contains electronic mailing lists.

The most important thing in the data base is the information about people, since this is what enables the rest of the network software to link a particular user ID on a particular machine with access privileges to data on all the other machines on the network. For example, the network file-transfer system uses the data base to decide which accounts on other hosts users can access automatically; a user can transfer files to his own account on another machine without having to log in to the other machine.

Since one person usually has accounts on several machines, it is useful to designate one account for receiving mail. One of the properties associated with a person in the data base is his preferred computer mailing address. The mail system accepts mail addressed to the person or to any of his computer accounts and sends the mail to the account where he prefers to receive it.

The FINGER program prints out a list of users currently on the system and some of the information the data base knows about them and where their terminals are. FINGER can also see whether a specific person is logged in to any of his accounts on the network or show all the public information about him such as his home phone number and his intercom number.

Our TN program permits I/O redirection across the network. It is also fully integrated with MUF, so that the user, once logged in to one machine, can log in to any other account on any machine with no more than one or two keystrokes.

Our data base is a novel implementation of a general-purpose "relational" data base in LISP. The data base runs as a server process on one of the DEC-20s and accepts requests for connection from other machines on the network. Users access the data base via the program UDB. UDB translates user requests into data-base transactions and opens a network connection to the data base; once a connection is established, the data base accepts modifications and requests for information. Access to the data base is controlled by a flexible protection scheme encoded in the data base itself.

For example, a user whose wife has just walked out on him wishes, of course, to update the data base so as not to be constantly reminded of her whenever the department issues invitations and announcements. He types "udb" and then, in response to the UDB prompt, "modify" and his name. UDB then displays each field of his data-base entry, and he enters a null in the Spouse field and exits. The data base considers him eligible to modify that information and updates itself. On the other hand, a new user who wishes to be added to the data base is certainly not eligible to do so. The data base will permit only a very few people to add or delete entries (as opposed to fields within entries).

So far this scheme has served us fairly well. Most users are responsible about updating their own information, greatly easing the task of maintaining the data base. When a large-scale change takes place -- the entire department's phone system just changed, for instance -- we update the data base en masse to keep it useful.

The server periodically writes several representations of frequently used relations into text files. These files are read-only and provide fast access to the data base. Hosts on the network periodically transfer these text files to their local file systems and make them available for use by : Thes programs. Speed has been traded off against strict accuracy text files become outdated as the real data base is modified. But changes made to the data base are distributed to each host on the network every fifteen minutes, so the delay is slight.

To make our network reliable we needed to add a small but important feature to the Chaosnet operating-system software. The basic problem is that server programs need a reliable way to determine who is at the other end of a network connection. To solve this problem, we modified Chaosnet to transmit user IDs and capabilities'securely over the connection. The server program uses this information, along with the user IDs stored in the data base, to determine who opened the connection. Of course this scheme has problems if some of the hosts on the network cannot implement this kind of security, so the data base knows whether each host can or can't. If it can't, then many of the features described earlier become unavailable; the file-transfer program, for instance, will no longer do automatic log-ins.

The Grapevine data base developed at Xerox PARC has many of the same goals as our data base. Grapevine differs from the Tools data base mainly in being distributed -- there are data-base servers on several computers. This distribution makes Grapevine a much larger system than the Tools data base. Grapevine is not a general data base, and it appears to be fairly difficult to add new kinds of information; the general, relational structure of the Tools data base makes it easy to add new kinds of data as we feel the need. Our data base was built in a few months of part-time work; Grapevine is in many ways more ambitious but it took many times longer to complete.

We are pleased with our current scheme and are considering several enhancements such as remote access to files and network-wide user groups.

8 Interactiveness versus Programmability

We have heavily emphasized interactiveness in the the Tools environment. The Z editor, the Session Manager, background compiling in Z, and the MUF process manager have all been designed to let the user accomplish common tasks with few keystrokes and little thought. The smaller Tools programs have been twiddled incessantly to get the best set of defaults for the majority of users. The user gets constant incremental feedback on the status of his computations. The goal has been to allow the user to concentrate on the task at hand without worrying about procedural details.

There are other environments, UNIX most prominent among them, that emphasize programmability [RITCHIE74]. In UNIX the shell, the operating system, the program library, and the C programming environment are carefully constructed to allow arbitrary composition of components. Folklore says that most UNIX programming is done in the shell, not in C. Each user can construct his own set of tools. UNIX is not, however, very interactive; even simple tasks require the composition of programs using pipes, I/O redirection, and shell scripts. The environment provides little feedback to the user as his tasks are executing [NORMAN81]. The user often has the sense of sitting at a remote job entry terminal, submitting batch processes, and waiting for them to terminate.

Consider the following task. A teaching assistant for a computer science course wishes, on the spur of the moment, to find out which of his students are currently logged on to the system and running a particular program. He wants to mail the results to the professor of the course.

On UNIX, the most likely approach would be to filter the output of the system-wide process status program PS through the pattern-matching program, GREP, and use a pipe to redirect the final output into a temporary file, then use the mail program to send the file to the professor. In the Tools environment, the most likely approach would be to let SM log the output of the system-wide process status program SYSTAT, edit the log, and then run the mail program in the background behind the editor.

Let's say that the teaching assistant is not intimately familiar with either UNIX or TOPS-20. The default for the UNIX program, PS, is to print out only his own process, not the processes of all users. The default for the TOPS-20 SYSTAT is to print out all the users, but not which program each user is running. So in both cases the assistant needs a reminder about an optional switch. On UNIX, for PS, he will do "help ps", which gives him the complete on-line documentation for PS:

```
    PS(1)               UNIX Programmer's Manual               PS(1)

    NAME
          ps - process status

    SYNOPSIS
          ps [ acegklstuvwx# ]

    DESCRIPTION
          Ps prints information about processes.  Normally, only your
          processes are candidates to be printed by ps; specifying a
          causes other users processes to be candidates to be printed;
          specifying x includes processes without control terminals in
          the candidate pool.

         All output formats include, for each process, the process id
         PID, control terminal of the process TT, cpu time used by
         the process TIME (this includes both user and system time),
         the state STAT of the process, and an indication of the COM-
         MAND which is running.  The state is given by a sequence of
         four letters, e.g. ``RWNA''.  The first letter indicates the
         runnability of the process: R for runnable processes, T for ...
```

... and so on for two hundred lines, almost all of which he has to read to discover that the form of the command he needs is "ps -axu". To get the same kind of help, the Tools-using assistant, taking advantage of TOPS-20's command-parsing facility, types

```
"systat ?"
```

and gets:

```
   One of the following:
    ALL         CLASS       CONTROLLING     DIRECTORY     HEADER
    LIMIT       LINE        LPT             MOST          NO
    PROGRAM     STATE       SYSTEM          TIME          WHAT
    WHERE       WHO
     or user name
     or directory name
     or decimal job number
```

```
or ","
or "."
or confirm with carriage return
```

... complete in eleven lines. The reminders are terse but mnemonic and rely on the interactiveness of the system for efficiency. If the assistant guesses wrong and types "systat program", as he probably will, no great harm is done -- the output from the incorrect command is null -- and he can guess "systat what" second time around.

Now the teaching assistant on UNIX will have to go through a whole set of steps that are obviated in the Tools environment. First he will have to run PS at least once (supposing that he gets the right command options straight off) to look at the format of its output. Then he must construct a regular expression for GREP to use in picking out the desired lines: "ps -axu | grep csh". Then (again supposing that he gets the right regular expression straight off) he must redirect the output to a file: "ps -axu | grep csh >csh-users". And finally, to be sure that no disasters have befallen the output, he must type out the file.

In the Tools environment, SM has automatically been logging the output from SYSTAT along with all the other transactions of the assistant's terminal session. A single keystroke takes him to the log. The power of the Z editor makes reading the output and picking out the desired information simultaneous: One two-keystroke command adds any lines he likes to a buffer while he first looks at them, so that he never need look a second time. Having identified all the users he was looking for, he dumps into a file the buffer containing their names.

To mail his file on UNIX, the teaching assistant must invoke the mail program, type in an escape sequence that indicates he wishes to mail a file, type in the name of his file, and then type in a command sequence that sends it. To mail his file in the Tools environment, he simply adds a "To:" line to the beginning of the file and uses a two-stroke command to run the mail program in "background compile" mode behind the editor, mailing the file instantaneously.

The chief characteristic of the Tools environment is its naturalness, the sense the user has of moving to the completion of a task in terms identical to his original conception of the task. The teaching assistant need not stop to think about the output of SYSTAT as a set of regular expressions; he need not stop to think about seeing what students are running his program as extracting a subset of those expressions and redirecting that subset to a file. He need not think procedurally at all. Instead he performs a series of small steps, seeing the results of each step appear instanteously on the screen. Any mistake he makes is necessarily trivial and quickly undone. The sensation of working in the UNIX environment is spiky, bracing, all or nothing. The sensation of working in Tools is smooth, thoughtless, casual.

Which is better, interactiveness or programmability? Obviously this example is biased in favor of interactiveness. There are many times when the programmability of UNIX is invaluable. If, for example, the teaching assistant wanted to repeat this same task over and over, he would have no choice but to reconceive it in programming terms. Often, however, UNIX requires the user to divert his attention from the primary task at hand and concentrate instead on programming. The Tools environment subordinates simple tasks to goals in a way that users find easy and natural.

The best of both worlds is always, trivially, a desirable goal. Accomplishing that goal is non-trivial, however, since interactiveness and programmability often conflict. The Tools group was effectively discouraged from exploring programmability on TOPS-20 because of the existence of a large amount of essential but "prehistoric" software that lacks the simple notions of standard I/O and command-line arguments. From the other end, we are encouraged to note that the people working on Berkeley UNIX are attempting to improve its interactiveness [xxxxxxx].

9 More Raw Materials: TOPS-20 as a Systems-Programming Environment

The normal environment for systems programming on TOPS-20 is typical of large time-sharing systems. The main programming language is assembly language. The DEC-10/20 instruction set is very powerful and well suited to certain applications (such as implementing LISP), but assembly-language programming remains a tedious chore.

There is no common set of subroutines to assist in developing programs. DEC supplies a primitive subroutine library, which consists mainly of assembler macros and error-handling routines. There are no routines for performing I/O (formatted or not), storage allocation, interrupt handling, or any other such applications. Some assembly-level user subroutine libraries have been developed (Columbia University's computing facility has developed a fairly extensive library), but they are still limited by the crudeness of the implementation language.

The quantity and quality of the comments on programs varies but usually fails to help anyone who didn't write a program understand it. And the opacity and confusion of assembly-language programs is well known. Frequently, a comment merely repeats the function of a statement without putting that function in the context of the surrounding code.

The development of the Tools systems-programming environment was motivated by inadequacies in the base TOPS-20 operating system and positive experiences with UNIX [RITCHIE74]. UNIX is one notable exception to the pattern of doing systems programming in assembly language and assuming that everybody else is going to do systems programming in assembly language.

The sophistication of the TOPS-20 operating system and the speed and large address space of the DEC-20 have made it possible for us to go far beyond UNIX in designing our environment. Compared with the PDP-11, on which UNIX evolved, TOPS-20 offers a much wider range of operating-system primitives for process and terminal control, I/O and file system operations, and memory management. We did not restrict ourselves to implementing run-time primitives and system utilities like those on UNIX --- it was not our intention to make a UNIX clone. We sought to exploit the full capabilities of TOPS-20. Towards this end, we implemented a large run-time library in a high-level language that provides a simple, powerful means to access almost all of the capabilities of the TOPS-20 operating system while hiding most low-level details.

10 The Tools Systems-Programming Environment

The Tools systems-programming environment is based on the belief that systems programming in general, and on TOPS-20 in particular, can and should be done in a high-level language. This belief was based in part on the success of C on UNIX. The belief is not radical. Computer scientists have recognized for some time the value of high-level programming languages. But in the "real world" (which again includes most of academy) the view is not dead that only assembly language can make systems programs both compatible and efficient. Thus our use of a high-level language is radical in practice.

The language of the Tools environment is BLISS [WULF71]. We use a slightly modified version of the BLISS-10 compiler developed at Carnegie-Mellon. BLISS is an Algol-like block-structured expression language whose distinguishing features are:

- Typelessness. BLISS has exactly one type: the machine word.
- Arbitrary access to "structures", i.e. records. A structure is defined by general expressions that compute addresses based on a pointer to an instance of the structure and arguments specified when reference to the structure is made.
- Uniform treatment of identifiers. The concepts "L-value" and "R-value" do not exist in BLISS. Identifiers always refer to addresses. A special unary operator (".") must be applied to an identifier to retrieve its value.
- Closeness to the machine. BLISS lets the programmer use low-level features of the machine architecture as he deems appropriate.

Had we attempted to implement the library in a strongly typed language such as Pascal, we could not have been nearly so successful. Strongly typed languages are not powerful enough to implement a run-time environment; they cannot manipulate types at run-time. So the designers of a typed language must implement both a run-time environment in the language itself and a set of run-time support routines in machine language. In contrast, BLISS presents no obstacles to implementing a run-time environment -- which is why its designers felt no obligation to package one with the language.

If a run-time library is to be effective, it must be easy to use. This means routines that take generic arguments and variable numbers of arguments, as many of the routines in the Tools run-time library do. To provide these conveniences in a strongly typed language requires that the programmer bypass the type system, resorting to such time-honored methods as external procedures and machine-language subroutines.

Another interesting comparison is between BLISS and the programming language C. Since UNIX provided our initial motivation, we discussed at considerable length the possibility of implementing a C compiler for the DEC-20 and using C as our language. We had two objections to using C. First, though there are a few C compilers for the DEC-20, they produce very poor object code compared to BLISS compilers. Second, many C programs depend, either blatantly or subtly, on memory organized in 8-bit bytes and 16- or 32-bit words. Converting trivial C/UNIX programs to run on the 36-bit word DEC-20 would not be hard, but trivial programs are also the easiest to rewrite in another language. Converting the complicated but more useful programs such as TROFF or AWK to run on a 36-bit machine would be difficult. Converting C/UNIX programs that were heavily dependent on the UNIX operating system, such as the shell and the directory manipulation tools, would be impossible; they would have to be completely reimplemented.

Another advantage of BLISS over C is its powerful data structuring. Unlike C, BLISS provides uniform reference to data structures [Geschke75]; it separates the syntax of structure references from the actual implementation. This lets the programmer change the implementation of a data structure without changing all the references to it. In C, there is no separation between syntax and implementation; when the implementation of a data structure changes, all references to it must be tracked down and changed, a formidable undertaking for large programs.

Much of the power of BLISS's data structuring derives from its clean pointer/value semantics [Wulf70, Wulf71]. In most languages, identifiers and expressions are interpreted differently according to whether they are used on the left-hand or the right-hand side of an assignment; in BLISS, expressions are interpreted without regard to context. In essence, all expressions are address (pointer) expressions; the unary operator "." must be used to extract the contents of an address. BLISS's clean semantics, combined with its uniformly referenced data structures, make structure definition easier and more efficient than in C. Many programmers, when they first encounter BLISS, balk at its unique semantics, especially at the ubiquitous ".". But like the parenthises of LISP, the "." has a power that experienced BLISS programmers find far outweighs any minor syntactic clumsiness.

We were not concerned with language portability. Our initial goal was to provide a high-level interface to all the features of the TOPS-20 operating system, features most other operating systems lack. Since BLISS is ideally suited for programming on the DEC-20, it was the logical choice.

If the choice of language was important, so were the primitives chosen for the run-time library. Developers of subroutine libraries often make the routines too low-level; library routines should be significantly less trivial than what a programmer would want to re-code himself. The Tools library contains many routines that are essentially "mini-interpreters". These routines take complex arguments and perform complex operations.

The major functions provided by the library are storage allocation, string manipulation, file i/o, formatted and unformatted I/O, software interrupt control, process control, and terminal control. Other more specialized routines exist for pattern-matching on regular expressions, manipulating symbol tables, network communication, sorting, priority queues, inter-process communication, and electronic mail.

Great Stuff!

As we worked, we learned that a systems-programming environment depends on social as well as technical matters. An ethos developed: a set of grup standards and a sense of common responsibility. Clean, consistent programming style, the sharing of code and experience, a willingness to make changes that require old programs to be modified -- these were as important to our success as the implementation language and the careful choice of primitives for the subroutine library.

Muddy, inefficient coding is possible in any programming language, structured or not. And by the same token, our experience has been that, given a modicum of self-control, clean, clear programming is also possible in any programming language. While there is wide belief that BLISS is particularly ill-suited to clean coding, we found it no worse a vehicle than any other Algol-like language.

The high coding standards of the Tools project are the result of peer pressure. It is the rule, not the exception, for programmers other than the author to make stylistic and functional changes in a program. During the course of the project at least ten people have programmed in the environment, and all the code anyone writes is subjected to perusal and review by some subset of those programmers. To counteract possessiveness about code, we have made a great effort to maintain an ethos of "egoless programming".

Sharing code and generalizing code to form routines for the run-time library also involve social issues. For a subroutine library to grow successfully, users of the library have to modify and contribute to it. Many parts of our library originated in specialized code from a particular tool; someone, either the original author or some other programmer, then generalized it to make it useful in a wider variety of applications. The commitment to making such generalizations, although expensive at the time, has saved time in the long run by allowing us to share each other's work.

Often after a large software project has developed and become stable for two or three years it becomes very difficult to make major changes and improvements. In the case of the Tools project, we have at least three times made major changes that required "rebuilding the world" -- examining and recompiling every single Tools program. Probably the most colorful of these was the decision to redefine the representation of the Boolean values TRUE and FALSE. There were at that point already several dozen Tools programs that had to be modified to accommodate the change. We have a commitment to making major changes that improve efficiency and overcome past structural and design limitations. While the transition is sometimes painful, the improvements pay off in the long run.

11 Comparison with Other Similar Projects

As we said, we began working on the Tools environment because of good experiences with UNIX. But UNIX is (as we've already indicated) by no means perfect.

C, the UNIX programming language, is highly regarded, but we would assert that the successes of the environment are due to clean coding practice and not to C per se, any more than our successes are due to BLISS. The failures of C and UNIX have resulted from forgetting these principles and placing faith in the idea that simply writing in C all by itself makes things clean and easy to understand. The UNIX environment contains numerous examples of code whose style falls below the high standards that the system aimed at.

Another problem with UNIX is that it has become stagnant. The designers of UNIX implementations on larger systems such as the VAX-11 have missed the opportunity to take advantage of increased resources. Rather than treating the transfer of the UNIX environment to more flexible machines as a chance to re-think design issues, the UNIX world has been satisfied to keep things static.

This stasis would be no problem if UNIX were the best possible environment for users and for systems programming on the larger machines. Unfortunately, it is not, and experience on both UNIX and TOPS-20 has made us realize the extent to which UNIX lacks the mechanisms necessary to build itself up and make itself more sophisticated. We believe that UNIX should be reconsidered in the context of the larger machines. There comes a time when compatibility with earlier versions must be subordinated to higher goals.

Three other projects that bear some similarity to our own are the multi-forking EXEC, BBN's MXEC [ASH81], and the programmable EXEC.

The multi-forking EXEC (MF-EXEC) is a version of the DEC EXEC with modifications that allow users to keep several active forks. It serves a function similar to MUF's, but the two differ greatly both in user interface and in the way they were engineered.

There are several user-level differences between the MF-EXEC and MUF. One difference is that in the MF-EXEC the user types the full name of a program to resume a fork; in MUF, the user types a single control key.

In MUF the user decides which programs he wants to keep. He specifies these programs in an initialization file. In the same initialization file, the user can specify parameters to be associated with a programs. For instance, he can specify the default arguments for a program. In the MF-EXEC, programmers managing the environment, not ordinary users, decide which programs will automatically be kept. We have found that users retain the same forks day in and day out. MUF lets the experienced user set up his environment exactly as he wants it; it's there waiting for him when he logs in, and he doesn't have to give verbose commands to recreate it.

Both as a utility for users and as a piece of software engineering, MUF has the advantage of being part of a large set of software tools with which it is well integrated. The same cannot be said of the MF-EXEC.

MUF acts as a better "fork changer" than the MF-EXEC. For example, in the MF-EXEC, to switch between a Lisp process and the editor, the user has to type "lisp" and "edit" and ^C and ^Z again and again. In MUF, thanks to well-engineered defaults, the same effect is achieved by single keystrokes, one to switch from the editor to Lisp, another to switch back.

In the MF-EXEC, arguments cannot be passed to a kept program. For example, on the first entry to the editor, the user can specify on the command line the name of the file he wants to edit. On re-entry, he cannot; he must resume the editor in the previous file and then give the editor command to change files. The MF-EXEC and the programs that run under it have no argument-passing convention. In the MUF/Tools environment, the user can resume a program either via a single control key, preserving all the arguments of the frozen state, or via a command line (reached by typing the control-space key first) which can contain new program arguments.

The engineering of the MF-EXEC is unimpressive. The implementation of the MF-EXEC required major modifications to the DEC EXEC, a 35,000-line assembly-language program. DEC has now at last included the modifications in its distributed sources, but disabled via assembly switches, and DEC does not support the disabled code. So any installation that chooses to use the MF-EXEC becomes responsible for re-integrating modifications and local bug fixes with future DEC releases.

At the time MUF was being developed, the MF-EXEC was not available. This was to our ultimate advantage. MUF now exists as a separate unit, written in BLISS and immune to the vagaries of DEC's software releasing procedures.

MXEC is similar in spirit to MF-EXEC with two major differences. First, MXEC has the ability to make macro extensions to the command language. Second, MXEC implements multi-forking with multiple jobs and "pseudo-terminals". The reason for this second feature is to have better control over the I/O flow of the multiple processes.

Our main objection to MXEC is that it is inappropriate to TOPS-20. The problem MXEC is trying to overcome by the use of multiple jobs is that multiple processes can write to the terminal simultaneously, their output being randomly intermixed; as the "valve" controlling the pseudo-terminals, MXEC can control output from each of the jobs. Unfortunately, sub-jobs and pseudo-terminals are very expensive on TOPS-20. If the goal is to produce usable, efficient software, MXEC fails. Using multiple processes that generate output to the terminal is impractical unless one also makes the commitment to high-resolution display terminals where each process can output to its own area of the screen.

In the Tools environment, SM, the session manager, deals with the problem of the terminal output of multiple processes. In fact, SM uses pseudo-terminals (but not sub-jobs). We wanted an efficient system and we were limited by our terminals, so we compromised. SM maintains the output of one process very well, and it can maintain the output of more than one process with MUF doing the synchronization.

The third project that somewhat resembles ours is the PCL-EXEC, which addresses the issue of interactiveness versus programmability by providing a shell programming language with an interface to the TOPS-20 command parser. The main advantage of the PCL-EXEC is that it allows the user to define new commands that take full advantage of the parser. For example, the syntax for the Tools pattern-matching program GREP looks like this:

```
GREP [-cnsvl99] pattern file [file ...]
```

The optional single-letter switches are not highly mnemonic -- the user must read the help file if he doesn't remember what they all do. But with the PCL-EXEC, it is possible to define "full" syntax for GREP that tells the user what the program expects at each point in the command, with switch names fully spelled out. By typing ?, the user can instantly find out the syntax of the GREP command.

```
GREP ?
 pattern, or one of the following:
 /ignore-case /no-text /straight-text /verbose /skip-
```

Of course, the TOPS-20 parser will also recognize abbreviations and complete them automatically, so that the terseness of the TOOLS/UNIX syntax style is not sacrificed.

The user defines new commands in an interpreted ALGOL-like language with built-in functions for accessing the parser and other parts of the operating system. This language is the main disadvantage of the PCL-EXEC. Though trivial for expert programmers to learn, the PCL-EXEC language really is beyond non-programmers. Defining simple commands requires a significant amount of programming, which violates our philosophy that simple tasks should be simple to express. For example, both the TOOLS environment and UNIX provide very simple methods for creating parameterized command scripts; the facility is used heavily by all experienced users. To accomplish an equivalent task using the PCL-EXEC requires a fair amount of programming, enough to discourage even programmers from using it often and of course excluding non-programmers completely.

The PLC-EXEC also has seemingly trivial engineering flaws that discourage its use in the Tools environment. When a Tools program is modified or a new one is added, it is "released" for public use merely by placing it in a special directory; users invoke the program merely by typing its name. But to invoke a PCL procedure, either it must be preloaded into the EXEC when the EXEC is built, or the user must explicitly ask for it to be loaded in his current EXEC. It is unreasonable to ask users to load PCL procedures. Not only does it place an unnecessary burden on them, it puts the operating system under stress because explicitly loaded procedures are not shared in memory like other programs -- each user has his own copy. On the other hand, preloading public PCL procedures into the EXEC is non-trivial; it forces a new EXEC to be rebuilt even for tiny changes to one PCL procedure, placing an unnecessary burden on the systems programmer and discouraging changes.

We would have liked to use the PCL-EXEC to define the command-line syntax of all the Tools programs. Each Tools program would have had a PCL procedure that took full advantage of the TOPS-20 command parser to parse its command line and then invoked the program with a syntactically correct command line. But because the PCL language is so verbose, and because it is too weak to support an embedded special-purpose language, we have stayed with our simple UNIX-style syntax. Manually writing and maintaining a 20-line PCL procedure for each of the 170 Tools programs would consume more time than we have.

Given that the PCL-EXEC is programmable only by expert programmers, do they find it a suitable tool? Only partially. Its language, like most shell languages, is very primitive; tasks that would be trivial in Lisp or APL are kludgy, difficult, or impossible in the PCL-EXEC. The only data types allowed are strings and numbers; there are no sequences or sets. Though the interface to the parser and the operating system is clean, it is incomplete, preventing the programmer from getting at many useful functions. A shell language should be either highly specialized, in order to make common tasks very easy (as on UNIX), or highly powerful, a full programming language that gives the expert full access to all the features of the system [LispShell?]. The PCL-EXEC is the inharmonious middle -- it isn't simple enough for simple tasks, and it isn't powerful enough for hard ones. (It is also a good example of how academic research doesn't pay attention to the minor engineering details that turn out to be so important in comfortable software.)

The features of the multi-forking EXEC, the PCL EXEC, and the Tools EXEC have recently been merged into a single Tools EXEC. Many of our occasional users have found the multi-forking features easy to learn, but experts still prefer MUF for its speed and succinctness. Only a few very experienced users are taking advantage of the programmability of the PCL EXEC. While we admire the efforts of the implementors of the various EXEC extensions (all of them from universities), it is staggering to realize the amount of extra work that went into maintaining such an ungainly assembly-language program.

12 The Tools Approach

Academic researchers, if they program at all, tend to implement prototypes that demonstrate only gross feasibility. The moment they can publish their results they move on, leaving their software behind them. Industry cannot afford to experiment much with computing environments except for the kept hands in a few prestige laboratories. The software that industry markets is expensive; customers demand that it be stable, that it be compatible with not only the features but the bugs of older software, and above all that it be on time. Anything that works at all is left untouched. So academy knows what to do but not how to do it, and industry knows how to do it just well enough but not any better than that. Nobody knows how to do it right.

Our aim has been to distill state-of-the-art concepts into simple, practical programs that will run on large, timeshared machines. The concepts in the Tools environment are not unique; what distinguishes Tools is that all these concepts have been smoothly integrated and implemented on a standard operating system, for heavily used machines. We were not interested in programs that could be run only on lightly loaded systems; this commitment to heavy usage would in itself be sufficient to distinguish the Tools project from most academic research on programming environments. Even when the machine is very heavily loaded, Z and MUF still give fairly good performance. We believe that the principles we have worked out in the Tools project offer good guidance for using timeshared computers as long as such machines continue to exist.

Much of the Tools project is not research at all by traditional standards. All of the major components and most of the smaller utilities have been done before, some of them much more powerfully. There are multiple-process managers more sophisticated than MUF; many screen editors have features that Z lacks; and it's easy to point to windowing systems more complicated than SM. Since we were at work right in the heart of academy, we heard over and over, "Stop that hacking and go do research." (This paper is in part a response to such chiding.)

But where else could something like the Tools project take place? Certainly not in the real world of production software. We compromised, but only where we chose to compromise, not where ignorant buyers could be persuaded to purchase whatever we produced. We never flinched from changing a program that we wanted changed. If the over-all design was bad, we redid it; if the implementation was sloppy, we rewrote it. When a better user interface was suggested, we started over. When useful new programs came along, either from within the Tools community or from outside, we modified the old programs or the new, or both, to make them work smoothly together. We devoted huge amounts of time to the thousands of little details that are individually trivial but that taken together determine the quality of a computing environment. It would be suicidal for the production sectors of industry to aim at anything better than a cheap, flashy suit, regardless of how it pulls and chafes. We had the determination and the energy and we took the time to make an environment that fits like a comfortable old sweater.

REFERENCES

@UndergraduateThesis(ANDERSON79, KEY "Anderson", AUTHOR "Anderson, Owen T.", TITLE "The Design and Implementation of a Display Oriented Editor Writing System", SCHOOL MITCSD, MONTH JAN, YEAR "1979")

@UndergraduateThesis(WEINREB79, KEY "Weinreb", AUTHOR "Weinreb, Daniel L.", TITLE "A Real-Time Display Oriented Editor for the Lisp Machine", SCHOOL MITCSD, MONTH JAN, YEAR "1979")

[Ash 81] Ash, William L.
Mxec: Parallel Processing with an Advanced Macro Facility.
Communications of the ACM 24(8):502-509, August, 1981.

[DEC 80] TOPS-20 Monitor Calls Reference Guide
Digital Equipment Corporation, Marlboro MA, 1980.

[Ellis 81] Ellis, John R.
Multiple Context Shells.
Research Report, YALECSD, January, 1981.

[Geschke 75] Geschke, Charles M. and James G. Mitchell.
On the Problem of Uniform References to Data Structures.
Technical Report CSL-75-1, XEROXPARC, January, 1975.

[Lantz 80] Lantz, Keith A.
Uniform Interfaces for Distributed Systems.
Technical Report TR63, Computer Science Department, University of Rochester, May, 1980.

[Metcalfe 76] Metcalfe, Robert M. and David R.Boggs.
Ethernet: Distributed Packet Switching for Local Computer Networks.
Communications of the ACM , July, 1976.

[Moon 81] Moon, David A.
Chaosnet.
AI Memo 628, Massachusetts Institute of Technology Artificial Intelligence Laboratory, June, 1981.

[Norman 81] Norman, Donald A.
The Trouble with UNIX.
Datamation 27(12), November, 1981.

[Ritchie 74] Ritchie, D. M. and K. Thompson.
The UNIX Time-Sharing System.
Communications of the ACM 17(7):365-375, July, 1974.

[Teitelman 77] Teitelman, Warren.
A Display Oriented Programmer's Assistant.
Technical Report SSL-79-9, XEROXPARC, March, 1977.

[Wood 81] Wood, Steven R.
Z: The 95% Program Editor.
In Proceedings of the ACM SIGPLAN SIGOA Symposium on Text Manipulation, pages 1-7. ACM, Portland, Oregon, June, 1981.

[Wulf 70] Wulf, William, et. al.
BLISS Reference Manual
1970.

[Wulf 71] Wulf, W. A., D. B. Russel and A. N. Haberman.
BLISS: A Language for Systems Programming.
Communications of the ACM 14(12):780-790, December, 1971.

Table of Contents

List of Figures