1700 PROGRAMMING

TRAINING MANUAL

CONTROL DATA INSTITUTE

INSTRUCTION INDEX

INSTRUCTIONS (by class)

Storage

Hex Code	Mnemonic	Section Number
1XXX	JMP	5.1.2.4
2XXX	MUI	5.1.2.2
3XXX	DVI	5.1.2.2
4XXX	STQ	5.1.2.1
5XXX	RTJ	5.1.2.4
6XXX	STA	5.1.2.1
7XXX	SPA	5.1.2.1
8XXX	ADD	5.1.2.2
9XXX	SUB	5.1.2.2
AXXX	AND	5.1.2.3
BXXX	EOR	5.1.2.3
CXXX	LDA	5.1.2.1
DXXX	RAO	5.1.2.5
EXXX	LDQ	5.1.2.1
FXXX	ADQ	5.1.2.2

Skip

Hex Code	Mnemonic	Section Number
010X	SAZ	5.2.1
011X	SAN	5.2.1
012X	SAP	5.2.1
013X	SAM	5.2.1
014X	SQZ	5.2.1
015X	SQN	5.2.1
016X	SQP	5.2.1
017X	SQM	5.2.1
018X	SWS	5.2.2
019X	SWN	5.2.2
01AX	SOV	5.2.3
01BX	SNO	5.2.3
01CX	SPE	5.2.4
01DX	SNP	5.2.4
01EX	SPF	5.2.4
01FX	SNF	5.2.4

Shifts**

Hex Code	Mnemonic	Section Number
0F2X	QRS	5.3
0F6X	LRS	5.3
0F4X	ARS	5.3
0FAX	QLS	5.3
0FCX	ALS	5.3
0FEX	LLS	5.3

Interregister

Hex Code	Mnemonic	Section Number
0808*	TRM	5.4
081X	TRQ	5.4
0818*	TRB	5.4
082X	TRA	5.4
0828*	AAM	5.4
083X	AAQ	5.4
0838*	AAB	5.4
084X	CLR	5.4
0848*	TCM	5.4
085X	TCQ	5.4
0858*	TCB	5.4
086X	TCA	5.4
0868*	EAM	5.4
087X	EAQ	5.4
0878*	EAB	5.4
08A8*	LAM	5.4
08BX	LAQ	5.4
08D8*	LAB	5.4
08E8*	CAM	5.4
08FX	CAQ	5.4
08F8*	CAB	5.4

Register Reference

Hex Code	Mnemonic	Section Number
00XX	SLS	5.5.5
02XX	INP	5.5.4
03XX	OUT	5.5.4
04XX	EIN	5.5.3
05XX	IIN	5.5.3
06XX	SPB	5.5.2
07XX	CPB	5.5.2
09XX	INA	5.5.1
0AXX	ENA	5.5.1
0BXX	NOP	5.5.5
0CXX	ENQ	5.5.1
0DXX	INQ	5.5.1
0EXX	EXI	5.5.3

PSEUDO OPS (by mnemonic)

Mnemonic	Section Number	Page Number
ADC	6.7	6-9
ADC*	6.8	6-10
ALF	6.9	6-11
BSS	6.12	6-15
BZS	6.12	6-15
COM	6.13	6-16
DAT	6.13	6-16
DEC	6.10	6-13
EIF	6.15	6-21
EJT	6.20	6-27
EMC	6.16	6-23
END	6.2	6-4
ENT	6.3	6-4
EQU	6.5	6-7
EXT	6.3	6-4
EXT*	6.6	6-6
IFA	6.15	6-21
IFC	6.18	6-25
LOC	6.17	6-24
LST	6.19	6-27
MAC	6.16	6-23
MON	6.22	6-29
NAM	6.1	6-3
NLS	6.19	6-27
NUM	6.6	6-9
OPT	6.21	6-28
ORG	6.14	6-19
ORG*	6.14	6-19
SPC	6.19	6-27
VFD	6.11	6-13

*Right most Hex number will include destination register.

**Third Hex number will include uppermost bit of shift count (bit 4).

1700 PROGRAMMING TRAINING MANUAL

THIRD EDITION

FOR TRAINING PURPOSES ONLY

This manual was compiled and written by instructional personnel of

CONTROL DATA INSTITUTE FOR ADVANCED TECHNOLOGY

CONTROL DATA CORPORATION

Publication Number 60207900B
February, 1970

The original draft of this manual was compiled and written by the Southern Region Training Staff. Technical revisions which have been incorporated in this printing were submitted by the Southern, Southeastern and Eastern Region Training Staffs.

Physical composition was accomplished by the Graphic Services Department within Control Data Educational Institutes. Since this department has continuation responsibilities for the originals of this manual, additional corrections, revisions, or suggestions should be submitted to the Manager of Graphic Services for processing.

TABLE OF CONTENTS

INDEX TO FIGURES

INDEX TO PROBLEMS*

*Note that solutions to problems appear in Appendix G.

PART I

1

CHAPTER I

1700 BASIC SYSTEM DESCRIPTION

CHAPTER I - 1700 Basic System Description

INTRODUCTION

The CONTROL DATA® 1704 Computer is a stored program, digital computer. Physically small, it is designed for high computation and input/output (I/O) speed. The program protection features of the 1704 Computer and high reliability under a wide range of environmental conditions make it suitable for real-time, on-line, or control applications.

The interface of the 1700 Computer System is capable of accepting a great variety of peripheral devices. Refer to Figure 1 for system characteristics.

Figure 1. 1700 Computer System Characteristics

Stored program, digital computer

Completely solid-state, 6000-type logic

Parallel modes of operation

18-bit storage word
- 16 data bits
- 1 parity bit
- 1 program protect bit

16-bit instruction word

Two 16-bit index registers

Multilevel indirect addressing

Magnetic core storage (options available):
- 4096 18-bit words, expandable to 32,768 words

Input/Output (options available):
- Transmission of 16-bit words or 8-bit characters

Console includes: Register contents displayed in binary; operating switches and indicators

Reliability (calculated):
- Approximately 8,000 hours mean time between failures for the 1704 Computer

Environment:
- 40° F to 120° F
- Relative humidity 0% to 80%

Cooling: Forced Air

System Interrupt

Flexible repertoire of instructions:
- Arithmetic operations
- Logical and masking operations
- Interregister transfers

Base 16 (hexadecimal) number system

Binary arithmetic:
- Modulus 2 - 1 (one's complement)

Intercomputer communications:
- 1700 to 1700
- Satellite operations

1.1 MEMORY

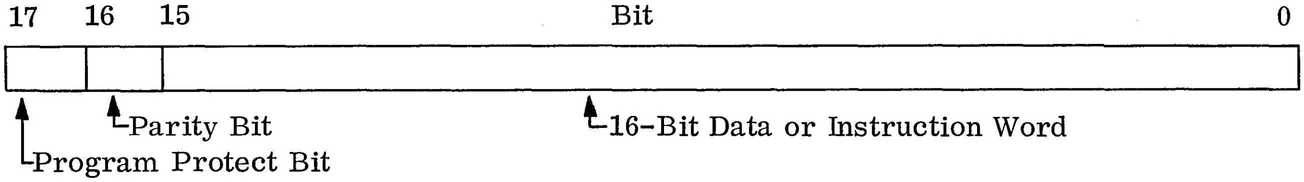
The basic 1700 Computer System provides high-speed, random-access magnetic core storage for 4,096 18-bit words. The storage capacity may be expanded from 4K by 4K increments to 32K as a maximum. With the addition of special hardware, memory may be doubled from 32K to 64K.

® Registered trademark of Control Data Corporation

1.1

Storage cycle time is 1.1 microseconds. This is defined as the shortest possible time between successive Read/Write operations in storage.

A storage word may be a 16-bit instruction, a 16-bit operand or a 16-bit address. A parity bit and a program protect bit are appended to each 16-bit storage word; thus a storage word is 18 bits long. Format:



Bit 16 is the parity bit. It takes on a value so that the total number of 1 bits is odd (total number of bits includes the program protect bit). For example, if all 16 data bits are 1's and the program protect bit is 0, the parity bit is a 1.

Bit 17 is the program protect bit. If it is a one, the word is protected and can only be modified or changed by a protected instruction.

1.2 PROTECT SYSTEM

The program protect system in the 1700 makes it possible to protect a program in the computer from any other non-protected program also in the computer. The combination of the high internal memory speed and the program protect system makes possible the use of the 1700 for background and foreground work. Foreground programs are protected and are generally multi-level (level 2 to 15) process programs. The foreground job is protected in core and runs at higher priority than background jobs which are assemblies, compilations, programs being debugged, etc. The background programs use the time available and are run in unprotected core. The protect system is enabled by setting the program protect switch on the programmer's panel. Any attempt to violate in any manner the protected portion of core from an unprotected instruction will cause a program protect violation which sets an interrupt on line 0 and also the program protect indicator which is visible as one of the fault lines on the programmer's panel. There are four program protect violations. They are:

1. An attempt is made by a non-protected instruction to write into a storage location containing a protected instruction or operand. It is legal to read from a protected area.

2. An attempt is made to write into a protected storage location by way of the external storage access when a non-protected instruction was the ultimate source of the attempt.

3. An attempt is made to execute a protected instruction following the execution of a non-protected instruction.

4. An attempt is made to execute interregister class instructions with bit 0 a one (M register is the destination); instructions EIN, IIN, EXI, SPB, or CPB. Later examination of these instructions will show how these are used to change the state of interrupts or the protected core area itself.

1.3 INTERRUPT SYSTEM

The basic computer (1704) provides two interrupt lines. Line 0 provides entry for interrupts generated as a result of a storage parity error, a program protect fault, or power failure. There are instructions available for the processing program to check for parity error or protect fault, and power failure can be assumed if one of the other two conditions does not exist. In the case of power failure, approximately 8 milliseconds of programming time are available; then the computer generates a master clear before the power actually goes down. Special hardware is available which can generate an automatic restart after the power comes back up.

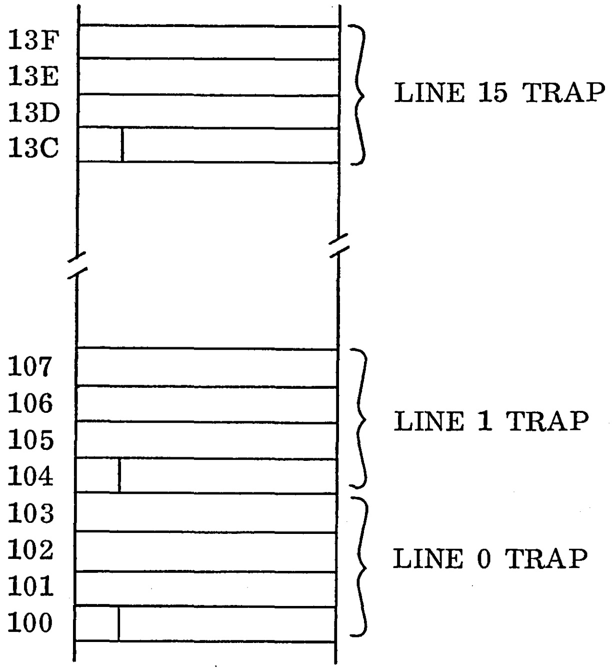
Line 1 provides for interrupts from the low-speed peripherals controlled by Equipment #1, the slow channel synchronizer. Any of the four stations (TTY, Card Reader, Paper Tape Reader, Paper Tape Punch) on the slow channel synchronizer sends its interrupt to line 1.

The 1705 provides expansion from two interrupt lines to 16 interrupt lines, to provide for additional equipment.

Interrupts are controlled by an interrupt mask register (M Register, 16 bits) which either allows selected interrupts in or blocks them out. Each line corresponds to its bit in the M Register. If the bit is a 0, any interrupt on that line is blocked out and must wait; if the bit is a 1, the interrupt is allowed in. The mask in the M register is set and changed under program control. Priority is established by the mask in the M register, not by the line position. In the case of concurrent interrupts on more than one line at the same priority, the lowest numbered line is recognized first.

There is a fixed group of core locations assigned to the interrupt system, locations $100-$13F, called the Interrupt Trap area. Four core locations are reserved for each line, beginning at $100 for line 0; each 4-word block is called the "trap" for that line.

Figure 2. 1700 Interrupt Trap Area



Contents of Trap:

word 3 - address of interrupt processor
word 2 - priority level for line
word 1 - RTJ to interrupt handler
word 0 - overflow and P

If the interrupt system is enabled and an interrupt occurs on a line that has a corresponding 1 in the M register, the hardware does the following:

1. disables the interrupt system (locks out all interrupts),
2. saves the contents of the P register (the address of the next instruction which would have been executed in the interrupted program) in the first word of the trap for that line,
3. saves the state of the overflow indicator (1 if set, 0 if not set) in bit 15 of that same word,
4. transfers control to the second word in the trap.

The above is <u>all</u> that the hardware does in handling an interrupt; anything else must be done by the software. Under most systems, the second word of the trap for each line is initialized by the software to contain a jump out of the trap to a routine (or routines) to save the registers of the interrupted program and handle the interrupt. The third and fourth words of each trap can be used by the software to contain anything desired; the standard operating system uses these words to hold the priority level of the line and the address of the processing program for the line, respectively. The interrupt processing routine may exit interrupt state (back to the interrupted program) through word 0 of the corresponding trap.

Example:

Assume line 1 has high priority, line 3 has lower priority. Line 0 always has highest priority. Any other running program has lower priority than either line 0, 1, or 3.

Make a table containing M register masks to be used while the routine servicing each line is running:

bit →	15															0	
MASKM	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	1	mask for main program
MASK3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	mask for line 3
MASK1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	mask for line 1
MASK0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	mask for line 0

Note that MASKM will allow all pertinent lines to interrupt; MASK3 will allow line 1 or line 0 to interrupt; MASK1 will allow only line 0 to interrupt; MASK0 will not allow any line to interrupt.

The processing programs would be set up as follows:

Main Program

Set M register to MASKM

Enable interrupt system

End

Interrupt Processor for Line 3

Store A, Q, I, M registers

Set M register to MASK3

Enable interrupt system

Inhibit interrupts

Restore registers

Exit interrupt state 03

Interrupt Processor for Line 1

Store A, Q, I, M registers

Set M register to MASK1

Enable interrupt system

Inhibit interrupts

Restore registers

Exit interrupt state 01

1.3

Interrupt Processor for Line 0

Store A, Q, I, M registers

Exit interrupt state 00

could just leave interrupts locked out during execution as shown here rather than set new mask (mask would be MASK0 if set and interrupts enabled)

Chapter 4 of the Computer Reference Manual contains more details about the Interrupt System.

1.4 INPUT/OUTPUT

Included with the 1704 is a slow channel synchronizer. This channel handles any or all of the slow speed devices normally affixed to a 1704. These are the 1713 Teletype, the 1729 Card Reader, the 1721 Paper Tape Reader, and the 1723 Paper Tape Punch.* Figure 2 shows a data pack extending from these slow speed devices through the slow channel synchronizer to both the A and Q registers. Transfer of information to these devices then is one word at a time (unbuffered) with the Q register containing address information and the A register containing data. The addition of a 1705 to the 1704 extends the A/Q unbuffered channel to eight more equipments. The 1705 also provides the addition of a direct access bus (DAC) to core. This provides buffered transfer of data directly to or from memory, bypassing the registers in the computer and, in fact, bypassing the normal compute channel of the computer. This direct access allows high speed transfer of data from peripheral devices like discs, drums, mag tapes, or high speed industrial equipment like multiplexers, etc.

1.5 BASIC SYSTEM DESCRIPTION

Figure 2 illustrates the basic structure of the 1704 and also extension to peripherals through the addition of a 1705. The basic 1704 is supplied with 4K (4096 words) of core storage. Core can be expanded in 4K increments to a maximum of 32K (or to 64K with the addition of special hardware).

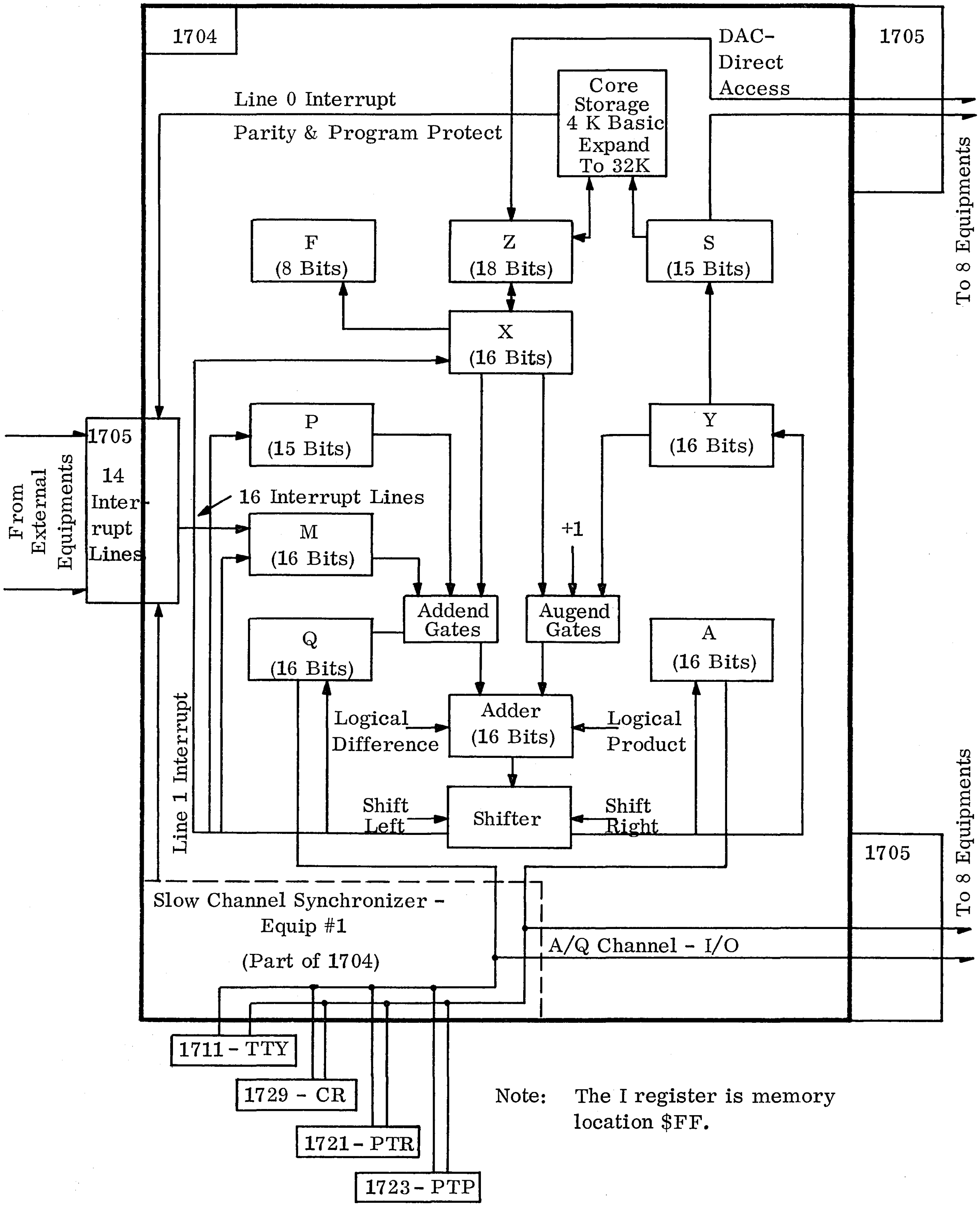
1.5.1 Registers

There are four registers that the programmer can get to directly from instructions. These registers are the A, Q, I, and M registers.

The A Register is the principal arithmetic register. It contains 16 bits (labeled bit-0 up to bit-15) of which bit-15 is the sign bit for arithmetic operations. The A register is also the register used to interface data during input/output operations to peripheral equipment.

*Also the variations of these (1711, 1713, 1722, 1724).

Figure 3. Basic System Description



Note: The I register is memory location $FF.

The Q Register is a multi-use register. Its uses include:

a. Auxiliary arithmetic register.
b. Retains part of the result of arithmetic operations such as multiply or divide.
c. Retains the most significant portion of the dividend during divide operations.
d. The Q Register is also used as the primary index register for address modification.
e. The Q Register supplies the addressing for peripheral equipments during unbuffered input/output operations.

The I Register is the second index register available. It is actually core location $00FF_{16}$.

The M Register or Mask Register controls interrupts. A one bit in any position of the mask register will enable an interrupt from the corresponding line number while a zero in any bit position of the mask register blocks the interrupt from the corresponding line number. The mask register is effective in controlling interrupts only when the interrupt system is enabled.

Other registers of interest to the programmer are:

P Register: This 15-bit register functions as the program address counter. It holds the address of each instruction, and after executing the instruction at address P, P is advanced to the address of the next instruction. The amount by which P is advanced is determined by the type of instruction being executed.

X Register: The X register is an exchange register containing 16 bits. This register holds data going to or from memory. It also holds one of the parameters in most arithmetic operations.

Y Register: The Y register is an address register containing 16 bits. It is in this register that storage addresses are formed and held for transfer during a storage reference.

The A, Q, M, X, Y and P Registers can be displayed and entered on the programmer's panel.

Shifter: The shifter is used by multiply, divide, and shift instructions. It is capable of shifting the output of the adder left and right one binary position or giving a direct transfer path to the arithmetic registers.

Adder: A 16-bit adder is used toperform allarithmetic andaddress calculations. Inputs to the adder are shown through the gates. The adder is a one's complement subtractive adder which is more fully described in the next chapter.

F Register: This 8-bit register is used by the control section of the 1704 for decoding instructions.

Z Register: This register communicates between the actual core storage and the computer through the X register. Notice this is an 18-bit register; the lower 16 bits are data or instructions from core, the 17th bit or bit #16 is the parity bit and the high order bit or bit #17 is the program protect bit. Core storage is described more fully in the next section.

S Register: This 15-bit register which is fed from the Y register is used to directly address core storage. Since core storage has a maximum of 32K locations, the S register need only be 15 bits.

The Z register and the S register are connected directly to external equipment through the 1705. This direct access channel allows insertion or extraction of core data directly to peripherals without program intervention.

Refer to Reference 1, Appendix A, for a more detailed description of the registers.

1.6 TYPICAL CONFIGURATION

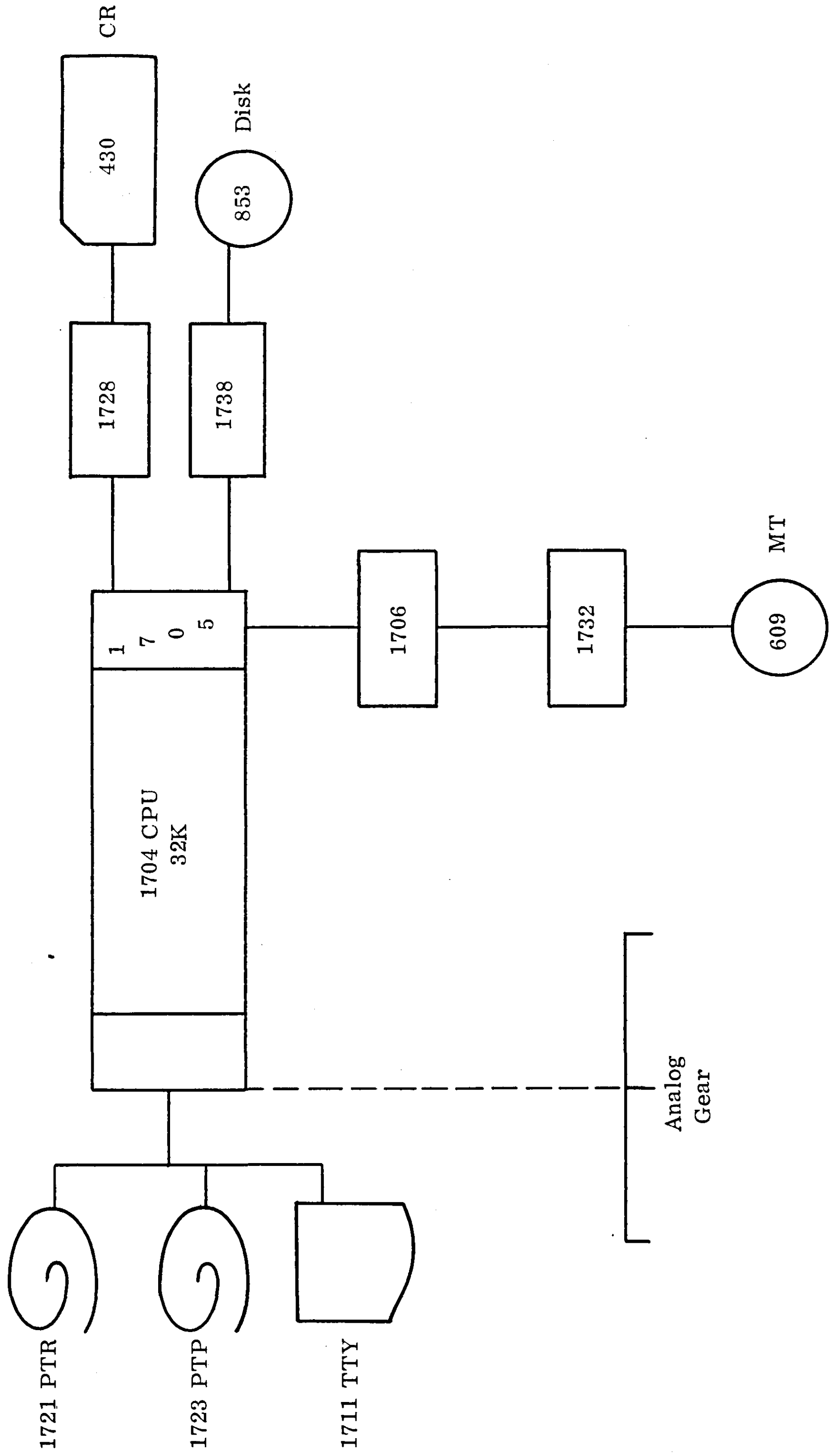


Figure 4. Typical Configuration

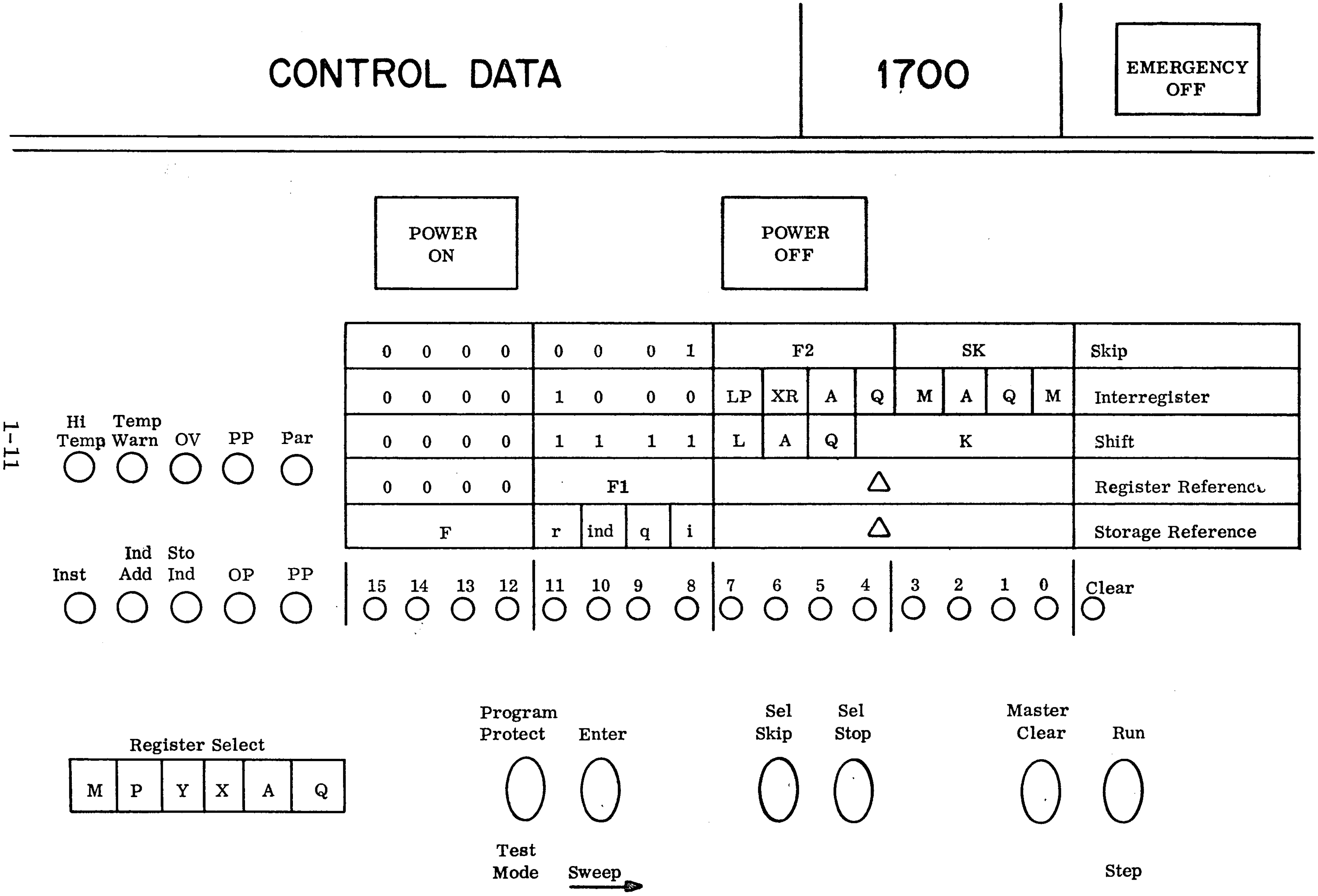


Figure 5. Console

1.7 CONSOLE DESCRIPTION

SWITCHES

Master Clear

This is a three-position key/lever switch. A Master Clear is executed whenever it is momentarily operated either up or down. A Master Clear returns the computer and peripheral devices to initial conditions by clearing all registers and peripheral equipment logic.

Run/Step

This is a three-position key/lever switch. When the switch is momentarily placed in the RUNposition, the computer begins program execution, starting with the instruction whose address is in the P register. The computer is stopped by momentarily placing the switch in the STEP position.

If the switch is repeatedly placed in the STEP position, the computer steps through the program, stopping after each storage reference. The significance of the storage reference just made is indicated by the Instruction Sequence indicators (INSTRUCTION, INDIRECT ADDRESS, etc.).

Enter/Sweep

This is a three-position key/lever switch maintained in all positions. The center position is off. Enter is used to enter memory; sweep is used to examine the contents of memory.

Enter

The ENTER position selects the Enter mode. In this mode, each Step operation of the RUN/STEP switch stores the contents of the X register at the location specified by P + 1 and then advances the P register by one. The first step after a Master Clear or clear P stores the contents of the X register at the location specified by P.

To store a few instructions in unprotected storage, proceed as follows:

1) Power is on but computer is stopped.

2) Operate Master Clear switch.

3) Press P REGISTER SELECT switch and CLEAR pushbutton, in that order. Set desired address for instruction in P by use of indicator pushbuttons.

4) Set ENTER/SWEEP switch to ENTER.

5) Press X REGISTER SELECT switch.

6) Press CLEAR pushbutton, then enter word to be stored by use of indicator pushbuttons.

7) Move RUN/STEP switch to STEP one time (carefully).

To store additional words in successive storage locations, repeat steps 6 and 7 until finished. To change to a new sequence of addresses, start at step 2 for the first one, then repeat steps 6 and 7 for each successive word.

A lighted indicator pushbutton indicates a "1", a dark one a "0".

Sweep

The SWEEP position selects the Sweep mode. In this mode, each operation of the RUN/STEP switch displays in the X register the contents of the storage location whose address is P + 1. The P register is advanced by one after each Step operation. The first step after a Master Clear or clear P displays the location specified by P. Instructions are not executed.

Selective Stop

This is a three-position key/lever switch. The computer stops when it executes a Selective Stop instruction if this switch is in either the up or down position. The up position is maintained; the down position is momentary.

Selective Skip

This is a three-position key/lever switch. Two Selective Skip instructions (SWS and SWN) are conditioned by this switch. This switch is off in the center position; the up position is maintained; the down position is momentary.

Program Protect/
Test Mode

This is a three-position key/lever switch maintained in all positions. The center position is off.

Program Protect

The PROGRAM PROTECT position selects program protection.

Test Mode

The TEST MODE position selects Test mode. This is used by the customer engineers for maintenance.

Emergency Off

Pressing this switch shuts off power for the entire system.

Register Select

The M, P, Y, X, A, and Q registers are available for display and manual entry of values via switch/indicators. A six-pushbutton switch/indicator, REGISTER SELECT, selects the register for display and entry.

Push the button for the desired register, and the contents of that register will light up in the 16-bit console binary register. A button lighted indicates a 1 bit; unlighted, a 0 bit.

If it is desired to change the contents of the register, push the CLEAR button (not Master Clear switch) to clear out that register. Then set the new contents in the register by pushing the button (it will light up when pushed) for each bit that should be a 1 bit in the register.

INDICATORS

Program Protect

The PROGRAM PROTECT bit indicator displays the state of the program protect bit of the last storage location referenced by the computer.

Faults

There are five fault indicators. When lighted, the fault condition is present.

- HI TEMP The temperature inside the computer has exceeded safe operating limits.
- TEMP WARN The ambient air temperature is approaching the maximum safe operating limit.
- OVERFLOW An arithmetic register overflow has occurred.
- PROGRAM PROTECT A violation of the program protect system has been detected.
- STORAGE PARITY A parity error has been detected in an operand or instruction read from storage.

Instruction Sequence Indicators

When an instruction is being stepped, this group of four indicators describe the meaning of the storage reference just completed. The data of the storage reference (read or write) is in the X register. The four indicators and their meaning when lighted are:

- INSTRUCTION: The contents of the X register is an instruction.

- INDIRECT ADDRESS: The contents of the X register is the result of indirect addressing. The indirect address may also be another indirect address, hence, this indicator may remain lighted for several consecutive storage references.
- STORAGE INDEX: The contents of the X register is the value of the Storage Index register.
- OPERAND: The contents of the X register is the value of the operand either written into or read from storage.

If more than one Instruction Sequence indicator is lighted, the computer is running. If only one indicator is lighted, the computer is not running or is in a rather unlikely program loop which does not use operands, the storage index, or indirect addressing.

CHAPTER II

1700 ARITHMETIC

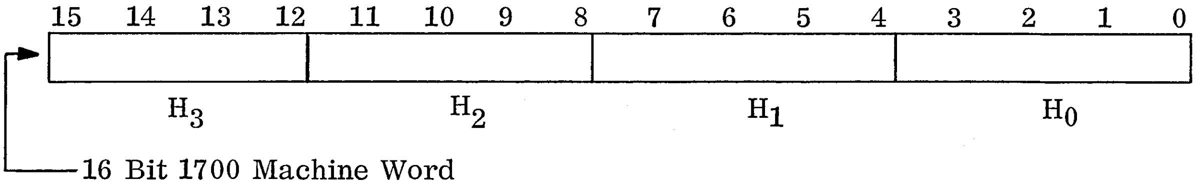
CHAPTER II - 1700 Arithmetic

Figure 6. Numbers

Decimal	Octal	Binary	Hexadecimal
0	0	0	0
1	1	1	1
2	2	10	2
3	3	11	3
4	4	100	4
5	5	101	5
6	6	110	6
7	7	111	7
8	10	1000	8
9	11	1001	9
10	12	1010	A
11	13	1011	B
12	14	1100	C
13	15	1101	D
14	16	1110	E
15	17	1111	F
16	20	10000	10

2.1 HEX-DEC CONVERSIONS

Since the 1700 is a 16-bit machine, it is convenient to group the 16 binary bits into 4 hexadecimal digits. This allows for quicker and easier manipulation of the arithmetic and easier identification of program dumps. The relationship of the 4 binary bits to each hexadecimal digit and the decimal equivalent is shown below:



The Range of Binary Bits in Each HEX Position Is:

Binary	HEX	Decimal	Binary	HEX	Decimal
0000	0	0	1000	8	8
0001	1	1	1001	9	9
0010	2	2	1010	A	10
0011	3	3	1011	B	11
0100	4	4	1100	C	12
0101	5	5	1101	D	13
0110	6	6	1110	E	14
0111	7	7	1111	F	15

This table must be memorized.

The arithmetic operations that are essentially basic involve binary to HEX, HEX to binary, hexadecimal to decimal, decimal to hexadecimal conversion, taking the one's complement of a hexadecimal number, and adding and subtracting hexadecimal numbers.

Each hexadecimal digit represents by its position in the number a certain power of the base 16. The least significant digit is a multiple of 16^0 which is 1; the second least significant digit represents a multiple of 16^1 which is 16; the third least significant digit represents a multiple of 16^2 which is 256; and the most significant hexadecimal digit represents a multiple of 16^3 which is 4096. It is necessary then only to do these multiplications followed by a final addition of the four multiples in order to convert a hexadecimal number to the decimal equivalent.

Example: Convert $13ED_{16}$ to Decimal.

$$\begin{array}{rcrcrcr}
D \times 16^0 & = & 13 \times & 1 & = & 13 \\
E \times 16^1 & = & 14 \times & 16 & = & 224 \\
3 \times 16^2 & = & 3 \times & 256 & = & 768 \\
1 \times 16^3 & = & 1 \times & 4096 & = & \underline{4096} \\
 & & & \text{Sum} & & 5101_{10}
\end{array}$$

There are many methods of converting a decimal number to hexadecimal. The simplest method involves successive divisions of the decimal number by 16. Each remainder becomes in turn the least significant hexadecimal digit of the converted answer, while each quotient becomes the next decimal number to be divided. Divisions continue until the quotient becomes zero.

Example: Convert 1476_{10} to HEX.

$$\begin{array}{rrl}
 & \underline{92} & \text{- Continue division} \\
16 & \big)\,1476 & \\
 & \underline{144} & \\
 & 36 & \\
 & \underline{32} & \\
 & 4 & \text{- First remainder is } H_0=4
\end{array}$$

```
       5 - Continue division
16  |92
     80
     12 - Second remainder is H1=C

      0 - Division stops
16  | 5
      0
      5 - Third remainder is H2=5
```

ANSWER - $1476_{10} = 5C4_{16}$

Addition of hexadecimal numbers is straightforward. Notice that 16 is carried from the least significant position to the most. The subtracting, of course, is the opposite of addition with 16 being borrowed from the most significant position.

Example: Add the HEX numbers 13CE and 2AA7.

```
  13CE
+ 2AA7
  ----
  3E75
  ↑↑↑↑
  ||||_________E (14) + 7 = 21 - Carry 16-Excess 5
  |||__________C (12) + A(10) +1 = 23 - Carry 16-Excess 7
  ||___________3      + A(10) +1 = 14 - No Carry-Excess E
  |____________1      + 2        =  3 - No Carry-Excess 3
```

As we'll encounter shortly, negative numbers in the 1700 are carried in one's complement form. The subtraction in each position of a digit from the largest possible digit in the base used yields the one's complement of the number.

Example: Find the One's Complement of the HEX Number 347E.

```
Subtract
FFFF
347E
----
CB81 - is One's Complement of 347E.
```

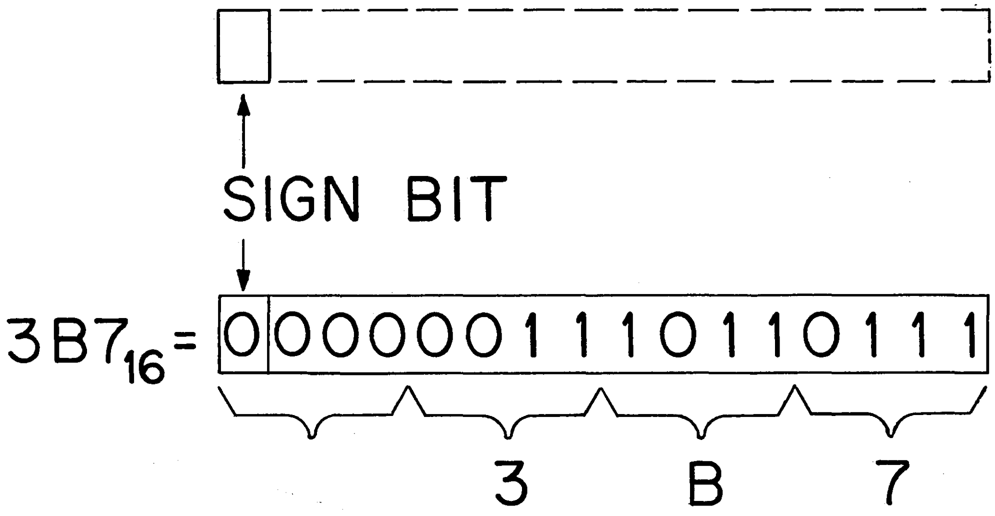
The same result can be obtained by converting both numbers to binary.

```
HEX       BINARY

FFFF =    1111 1111 1111 1111
347E =    0011 0100 0111 1110◄──────────────────────┐
CB81 =    1100 1011 1000 0001 ── is One's Complement of ──┘
```

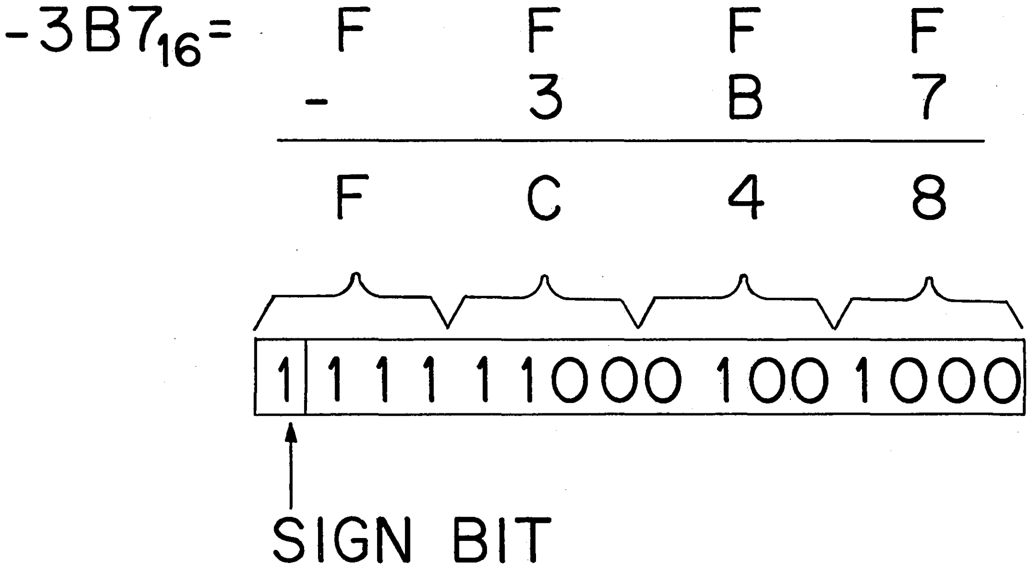
Notice that the one's complement of a binary number has all ones in the binary number changed to zeroes, and all zeroes changed to ones, which suggests another method of obtaining the one's complement of a number.

Figure 7. 1700 Integer Numbers

Example:

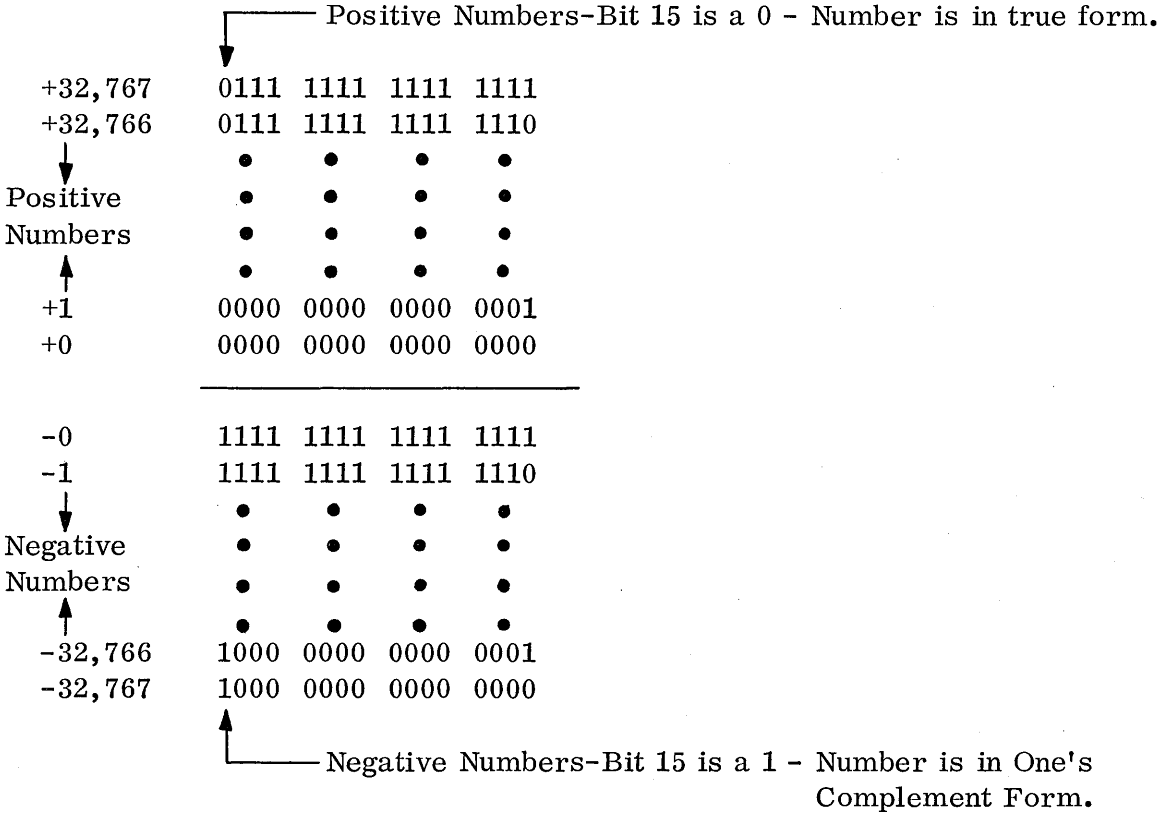


Negative Numbers Stored in One's Complement Form:



2.2 RANGE OF NUMBERS

Figure 8 illustrates the range of numbers used in the 1700. The total number of bits available is 16 and for arithmetic operations bit #15 will be a zero for positive numbers, and a one for negative numbers. For positive numbers, bits 0-14 are in true form. However, for negative numbers bits 0-14 are in one's complement form. Notice there are two 0's, a positive zero and a negative zero.



```
                 Positive Numbers-Bit 15 is a 0 - Number is in true form.

+32,767     0111 1111 1111 1111
+32,766     0111 1111 1111 1110
               •    •    •    •
Positive       •    •    •    •
Numbers        •    •    •    •
               •    •    •    •
+1          0000 0000 0000 0001
+0          0000 0000 0000 0000

-0          1111 1111 1111 1111
-1          1111 1111 1111 1110
               •    •    •    •
Negative       •    •    •    •
Numbers        •    •    •    •
               •    •    •    •
-32,766     1000 0000 0000 0001
-32,767     1000 0000 0000 0000

                 Negative Numbers-Bit 15 is a 1 - Number is in One's
                                                  Complement Form.
```

Note: Each Negative Number is Represented as the One's Complement of its Corresponding Positive Number.

Figure 8. Range of Numbers Used in the 1700

2.3 ADDER

A straightforward adder would pose problems in the 1700 in that negative zero in many cases would be produced as a result instead of a positive zero. Consider the addition:

```
 4321
+BCDE
 FFFF - Straightforward addition of the positive number 4321 and a negative num-
        ber of the same size, BCDE, yields a correct result of zero, but nega-
        tive zero instead of positive zero.
```

Since the skip tests do not recognize negative zero, the adder in the 1700, which is a 16-bit one's complement subtractive adder, eliminates minus zero in all but one case. It functions by taking the one's complement of the addend and subtracting this from the augend. The same addition as in the above is accomplished:

```
Add

 4321                           4321                          4321
+ BCDE ——► One's Complement ——► 4321 ——► Now Subtract          4321
                                         Same Result ———►      0000
                                         But +0
```

The one case where a minus zero is produced is the case where minus zero is added to minus zero.

```
Add              Subtract

 FFFF            FFFF
+ FFFF ————————► 0000
                 FFFF      (-0) + (-0) = (-0)
```

Negative zero can be converted to positive zero by simply adding positive zero to it.

```
Add              Subtract

 FFFF            FFFF
+ 0000 ————————► FFFF
                 0000      (-0) + (+0) = (+0)
```

For subtraction, the subtrahend is not complemented, but is subtracted directly from the minuend. The only case producing negative zero is (-0) - (+0) = (-0).

```
Subtract Directly

FFFF
0000
FFFF
```

Page 3-20 of Reference 1, Appendix A, expounds further on conditions causing negative zero.

2.4 OVERFLOW

Overflow in the 1700 is essentially a condition wherein the result of some arithmetic operation is too large to fit into its designated register. Overflow can be caused by add type operations, by a subtract type operation and by divide. In a one's complement computer like the 1700, overflow, when it occurs, sets the overflow indicator which remains on until tested. Refer to Figure 8. Notice that overflow will not occur when a positive number is added to a negative number since the result will always lie within the range of numbers. Adding two positive numbers together or adding two negative numbers together, or the equivalent operation (subtracting a negative number from a positive number or subtracting a positive number from a negative number), can cause a result which is too large to be contained in the A or Q Register. Recovery can be made from add type overflow operations and, in fact, overflow is often a useful tool for accumulating single precision numbers into double precision numbers.

Example: Add 1 to the Largest Possible Number

```
7FFF
0001 ──► Complement          7FFF ──── Note:  If the adder must borrow from
         Then Subtract ────► FFFE              beyond the most significant
                          ▲  8001              position, it subtracts 1 from
         Subtract Borrow  └────1               the result.
                             ────
                             8000 ──► Answer is now (-7FFF) which is wrong.
```

The computer logic will set the overflow indicator when both signs are initially the same and a borrow occurs.

Recovery can be made, however, by accumulating the overflow in a second cell and masking out bit-15 of the first, since bits 0 through 14 remain correct.

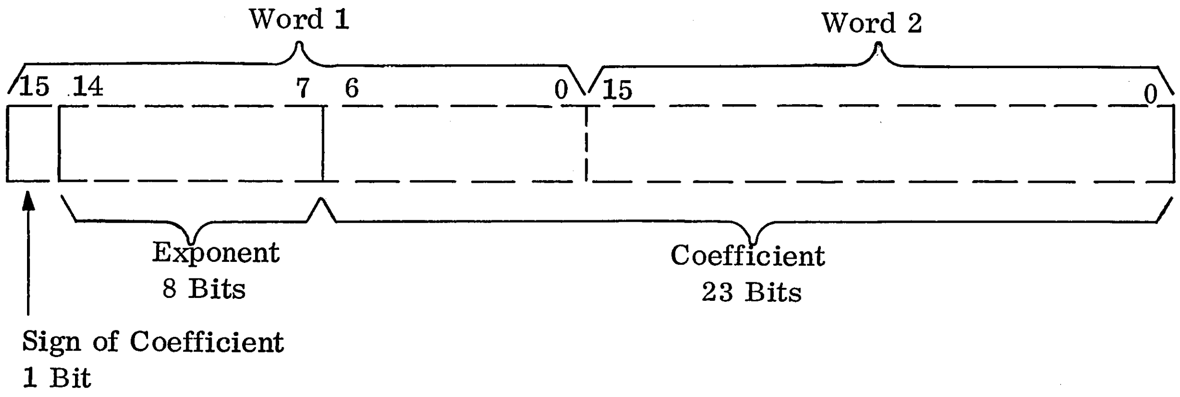
Remember that any add type instruction can cause overflow. This includes ADD, SUB, RAO, AAQ, etc.

Divide overflow will be described in more detail in the storage reference class of instructions for the DVI instruction itself.

2.5 FLOATING POINT NUMBERS

Floating point numbers will be of interest to scientific programmers, and since the Fortran compiler provides for them, their format will be discussed here. The range is $.591 \times 10^{-39}$ through 1.694×10^{39}.

Floating point numbers require two words of core and include: one bit, sign of coefficient; 8 bits, biased exponent; 23 bits, normalized coefficient.



A number, for example 25., would be packed as follows:

1. convert decimal number to hexadecimal

   $25._{10} = 19._{16}$

2. convert hexadecimal number to binary

   $19._{16} = 0001\ 1001._2$

3. normalize the binary number (move the binary point to the left of the first one bit)

   $0001\ \ 1001._2 = \underset{\text{coefficient}}{.11001_2} \text{ x } \underset{\substack{\text{exponent}\\\text{(power of 2)}}}{2^5}$

4. bias the exponent. In the 1700, all exponents (positive or negative) are biased by 80_{16} (80 is added to the exponent)

   $2^5 = 5 + 80 = 85_{16}$

5. pack the number in the two words:

   $$\underset{\text{sign of coefficient}}{\boxed{0}}\underbrace{\boxed{10000101}}_{\substack{\text{exponent}\\85}}\underbrace{\boxed{1100100\,\vdots\,0000000000000000}}_{\substack{\text{Coefficient}\\\text{Left Justified}}}$$

   $= 42E40000_{16}$

   A negative number would be packed as though it was positive, then both words would be complemented.

   $-19_{16} =$

   $$\boxed{1}\boxed{01111010}\boxed{0011011\,\vdots\,1111111111111111}$$

   $= BD1BFFFF_{16}$

   To unpack numbers (i.e., from a HEX dump) the reverse procedure is followed.

CHAPTER III

ASSEMBLY SOURCE FORMAT

Figure 9. Floating Point Example

EXAMPLE: PACK 375_{10}

$375_{10} = 177_{16}$

$16\,\overline{)375}$

$\overline{)23}$ r 7

$\overline{)1}$ r 7

0 r 1 ↑

$177_{16} = 0001\ 0111\ 0111_2 =$

$.1011\ 1011\ 1000 \times 2^9$

| 0 | 10001001 | 1011101 | 11000000000000000 |

$\underbrace{}_{4}\ \underbrace{}_{4}\ \underbrace{}_{D}\ \underbrace{}_{D}\ \underbrace{}_{C}\ \underbrace{}_{0}\ \underbrace{}_{0}\ \underbrace{}_{0}$

$$\begin{array}{r} 9 \\ +80 \\ \hline 89_{16} \end{array}$$

EXPONENT: 10001001_2

-375_{10} WOULD BE STORED AS

$$\begin{array}{rcccccccc} & F & F & F & F & F & F & F & F \\ - & 4 & 4 & D & D & C & 0 & 0 & 0 \\ \hline & B & B & 2 & 2 & 3 & F & F & F_{16} \end{array}$$

| 1 | 01110110 | 0100010 | 00111111111111111 |

2.6 EXERCISES

1. Group these 16 bit 1700 binary numbers into hexadecimal, convert the answers to decimal, then add the three together.

   a. 0010 1101 1010 1110

   b. 1000 1111 1100 0111

   c. 1111 1111 1100 0000

2. Convert these decimal numbers to HEX, and represent each in binary as they would appear as a 16-bit 1700 word.

   a. 4095

   b. -17

   c. 255

3. How would a signed number, either positive or negative, that occupied only an 8-bit field be expanded to occupy a 16-bit field?

CHAPTER III - Assembly Source Format

3.1 PROGRAM FLOW

Two phases are involved in processing the assembly language program. The first phase involves reading of the source program (the program prepared by the programmer) into the computer under control of the assembler program. The assembler program reads and decodes the instructions outputting a listing with the object program. The object program, which can be output on either paper tape, mag tape, or disk is in a formatted form suitable for loading back into the computer under control of the relocating linking loader program. This re-entry of the objectprogram into the computer through the loader is Phase Two. After loading, relocating and linking this program appropriately to other programs, the program can now be executed. Two large standard software packages, then, are involved in the assembly process; the assembler itself and the relocating linking loader. Figure 10 illustrates this diagrammatically.

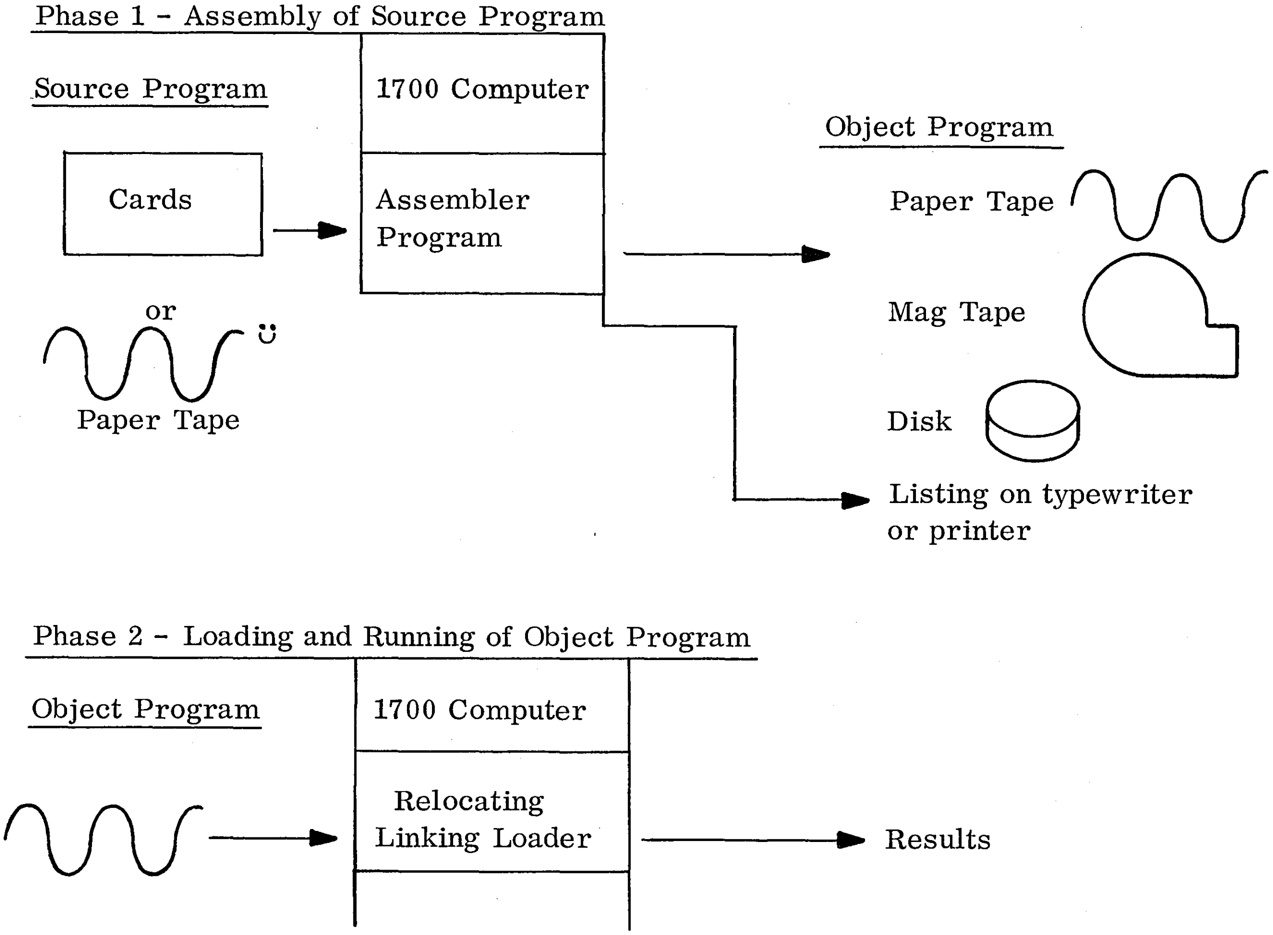

Phase 1 - Assembly of Source Program
Source Program
1700 Computer
Object Program
Cards
Assembler Program
Paper Tape
or
Mag Tape
Paper Tape
Disk
Listing on typewriter or printer
Phase 2 - Loading and Running of Object Program
Object Program
1700 Computer
Relocating Linking Loader
Results

Figure 10. Processing the Assembly Language Program

3.2 1700 ASSEMBLY LANGUAGE SOURCE FORMAT

Regardless of the standard assembler being used (utility assembler or macro assembler) the source format is prepared the same. This source format consists of four fields, the location field, the opcode field, the address field and the comment field as illustrated below:

Location	Opcode	Address	Comments

The total width of all four fields combined is 72 columns. Each field, however, can be of any length and the statements are said to be free-field. To signal the end of these fields, either a blank or a tab is used. The blank is technically used as the field terminator for card input source with consecutive blanks ignored and the first non-blank character signifying the beginning of the next field. For paper tape input source the tabular key depression is normally the field terminator since the source can be typed using tab fields to arrange the source type in an orderly fashion. Blanks, however, can be used for paper tape input source as field terminators also. Consecutive tabs will indicate the absence of a field. The carriage return key depression will signify the end of a statement for paper tape input source. The end of the card itself for card input source is the statement terminator.

3.2.1 Location Field

The location field is used for placing symbols which will define positions in the program. These symbols have a maximum length of six characters (more than six will not be processed by the assembler) and the first character must be alphabetic.

Example:

Location	Opcode	Address	Comments
A123	---	--- →	Legal
1A23	---	--- →	Illegal - must begin with alphabetic
TAGTOMB	---	--- →	Legal - only TAGTOM is processed.

The same symbol cannot be used twice in the same program, as this would constitute a doubly defined symbol, an error condition. Upon finding a symbol in the location field, the assembler places this symbol with its location in its internal symbol dictionary. Further references to this symbol then will yield its location.

An asterisk in the first column of the location field indicates that all subsequent information in that statement is to be treated as a remark.

Example:

```
Location          Opcode          Address          Comments

*  THESE LINES WILL BE PROCESSED AS A REMARK AND
*  APPEAR ONLY ON THE LISTING.  THEY DO NOT GENERATE
*  ANY MACHINE CODE.

START               --              --               --
```

Symbols appearing in the location field corresponding to certain pseudo instructions are meaningless. These will be discussed under the section dealing with the pseudo instructions. A numeric entry into the location field is allowed for one pseudo instruction only, the NAM pseudo instruction.

3.2.2 Opcode Field

In this field, machine instruction mnemonics or pseudo instructions or macro names (if the macro assembler is used) are placed. The machine mnemonics will be decoded by the assembler with the appropriate machine codes generated. The pseudo instruction will produce action by the assembler and may not, for some pseudo instructions, generate any specific machine instructions. Macro names will be discussed in Chapter VII.

Example:

```
Location          Opcode          Address          Comments

                   NAM               |             NAM is a pseudo-
                    |                |             instruction
                    |                |
START              LDA               |             LDA is a machine
  |                 |                |             mnemonic
  |                 |                |
  |                 |                |
```

3.2.3 Address Field

The operands used in the address field are:

Symbolic
Numeric
Asterisk
Special Characters (A, Q, M, 0, I, B)
Combination of above special characters joined by arithmetic operators (address expression)
Null (absence of operand)

Symbols used in the address field either alone or in an expression, must be legally defined. Besides appearing in the location field, symbols can be defined as being names in the address field of certain pseudo instructions.

3.2.3

Numeric operands in the address field can be either decimal or hexadecimal. To distinguish between the two, a $ sign would precede the hexadecimal number (1234 is a decimal, $1234 would be hexadecimal). The range of decimal numbers must be ± 32,767. The range for hexadecimal numbers is ± 7FFF. Expressions are formed by the combination of either symbolic or numeric operands with addition, subtraction, multiplication or division operators (+, -, *, /). Nesting is not allowed. The expression is scanned left to right with divisions and multiplications done first and a second scan left to right for addition or subtraction. An expression evaluated as a constant in the address field may be used only with the =X form of constant, not =N.

Example:

Location	Opcode	Address	Comments
TAG1			
TAG2	LDA	TAG1+6*$1C/4	

If TAG1 is at location 103_{16}, the expression is evaluated:

1st) 6 x 28 = 168

2nd) 168 ÷ 4 = 42 = $2A

3rd) 0103 + 2A = 012D, then the contents of core location $012D_{16}$ is loaded into the A register at run time.

The asterisk can be used in the address field to also specify the current location of the program counter when the instruction is assembled. If the instruction is two words long, the asterisk specifies the first word of the instruction. Even though the asterisk is also used in the address field as a multiplication sign, the logical use of the asterisk for both processes will not conflict.

Example:

Location	Opcode	Address	Comments
TAG	LDA	*-2	Will load A with contents of the core location 2 before TAG.
TAG1	LDA	**2	The first * refers to the location of TAG1, the second * is for multiplication.

The special symbols Q, I and B are used with the storage reference instructions to refer to index registers. Q refers to Q register index modification, I refers to the contents of location FF to be used as an index register, and B would specify both the Q register and location FF to be added to the base address to form the effective address.

Example:

Location	Opcode	Address	Comments
	LDA	TAG, I	The contents of core location FF are added to TAG to produce the effective address.
	LDA	TAG, Q	Same as above, but Q register is used.
	LDA	TAG, B	Both Q and I are added.

The address field for any of the interregister instructions requires either A, Q or M registers as a destination.

Example:

Location	Opcode	Address	Comments
	AAQ	A, Q	Add A to Q and put results in A and Q
	TRA	A, Q, M	Transfer A to A, Q and M registers

For the interregister instructions, A, Q and M, refer to the registers A, Q and M.

3.2.4 Comment Field

This field is used for remarks that are printed as part of the list output. Entries in this field do not produce any machine code.

If it is desired to put a comment on an instruction which does not have an address field (i.e., SLS) it is advisable to put a 0 in the address field, before the comment begins, to eliminate an assembly error message.

```
i.e.,  SLS       COMMENT      incorrect
       SLS   0   COMMENT      correct
```

3.3 ASSEMBLY LISTING

The assembly list consists of 18 columns of descriptive information related to the source statement, followed by a maximum of 80 columns listing the source statement.

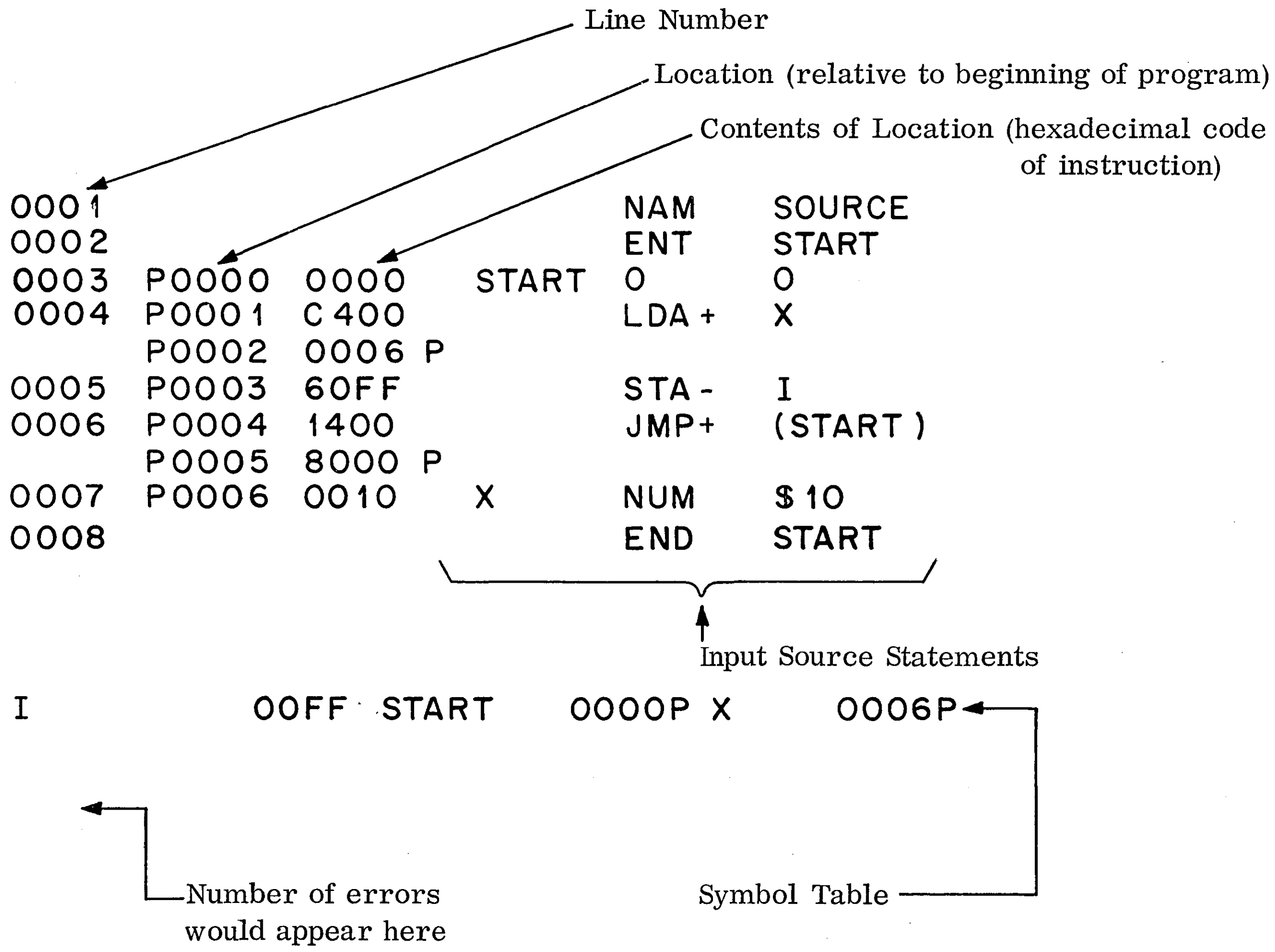
Column	Contents
1-4	line number; truncated to 4 decimal digits
5	space
6	relocation designator for location
	P program relocation
	D data relocation
7-10	location in hexadecimal
11	space
12-15	machine word in hexadecimal
16-17	relocation designator for word
	P program relocation
	-P negative program relocation
	C common relocation
	-C negative common relocation
	D data relocation
	-D negative data relocation
	X external
	blank absolute
18	space
19-98	input source statement

SYMBOL TABLE A table containing the location symbols, locations, and relocation values is printed at the end of pass 3 if the L option is selected. Format of the symbol table:

Column	Contents
1-6	symbol name
9-12	location
13	relocation of location
15-20	symbol name
23-26	location
27	relocation of location
29-34	symbol name
37-40	location
41	relocation of location

The columns not specified above contain spaces.

Figure 11. Sample Assembly Listing



```
0001                             NAM    SOURCE
0002                             ENT    START
0003  P0000  0000     START      0      0
0004  P0001  C400                LDA+   X
      P0002  0006 P
0005  P0003  60FF                STA-   I
0006  P0004  1400                JMP+   (START)
      P0005  8000 P
0007  P0006  0010     X          NUM    $10
0008                             END    START

I            00FF  START    0000P X      0006P
```

Line Number
Location (relative to beginning of program)
Contents of Location (hexadecimal code of instruction)
Input Source Statements
Number of errors would appear here
Symbol Table

Appendix D contains error messages.

4

CHAPTER IV

BASIC 1700 INSTRUCTION FAMILIARIZATION

CHAPTER IV - Basic 1700 Instruction Familiarization

4.1 LDA, STA, ADD ASSUMING DATA AVAILABLE

The Load A (LDA) instruction replaces the contents of the A Register with the contents of the referenced memory location. The contents of the memory location is not changed.

The Store A (STA) instruction replaces the contents of a referenced memory location with the contents of the A Register. The contents of the A Register is not changed.

The Add to A (ADD) instruction forms a 16-bit sum of the contents of the A Register and the contents of the referenced memory location and places this sum in the A Register. The contents of the memory location is not changed.

Problem: Replace the contents of location SAVE with the sum of the contents of locations SAVE and DATA, assuming that both locations contain legal data.

```
Location        Opcode      Address

                LDA         DATA
                ADD         SAVE
                STA         SAVE
```

4.2 LDA, STA, ADD USING PRESET DATA

The NUM instruction creates a table of constants listed in the address field behind the instruction. If a Label is given, it is assigned to the first value.

The BSS pseudo instruction reserves a segment of core to be used for any purpose. The data contained is unknown at load time.

The BZS pseudo instruction reserves a segment of core to be used for any purpose and fills this area of core with all zeroes.

Problem: Add zero to the contents of location TAG and store the result in location TAG2. Locations TAG and TAG2 should then be equal.

```
Location        Opcode      Address

TAG             NUM         $423
                BZS         TAG1(1)
                BSS         TAG2(1)
                -
                -
BEGIN           LDA         TAG
                ADD         TAG1
                STA         TAG2
```

4.3 LDA, STA, ADD USING PRESET DATA TO ILLUSTRATE LOOPING

The Skip if A is Zero (SAZ) instruction checks the contents of the A register. If A is all zeros, the program skips to a prestated location up to 16 locations forward, (never backward). If not, program control goes to P + 1 (the next location).

The Skip if A is Positive (SAP) instruction checks the uppermost bit of the A register. If this bit is "0" (positive), program execution will skip to the specified location up to 16 locations forward, (never backward). If not, the program will continue at P + 1.

The Skip if A is Minus (SAM) operates identically to the SAP instruction except that the uppermost bit of the A Register is checked for a "1", indicating a negative quantity.

Problem: Add 21_{16} to each of the quantities in array TAG1 and then store these new numbers at array TAG2.

Index Register I will be used for controlling the loop. It starts at 4 and progresses through 3, 2, 1, 0, then the loop ends when it turns negative.

Solution:

```
Location        Opcode      Address                Comments

TAG1            NUM         $4152,$0431,$0210
                NUM         $12F3,$F201
                BSS         TAG2(5)
TAG3            NUM         $21
ONE             NUM         $0001
SAVE            NUM         $0004
                -
                -
BEGIN           LDA         SAVE
                STA         $FF                    4→FF for loop control
                LDA         TAG3                   1st no. ——→ A
LOOP            ADD         TAG1,I                 (A)+(TAG1 +(FF)) ——→ A
                STA         TAG2,I                 (A) ———→ TAG2+(FF)
                LDA         $FF
                SUB         ONE                    Subtracts 1 from A
                STA         $FF
                SAZ         DONE-*-1               Skip to DONE if (A) = 0
                JMP         LOOP
DONE            SLS         0                      Stop Instruction
                END
```

4.4 STQ AND MUI INSTRUCTIONS

The Store Q (STQ) instruction replaces the contents of the referenced memory location with the contents of the Q Register. The contents of Q are unchanged.

The Multiply (MUI) instruction forms a 32-bit product of the contents of A (multiplier) and the contents of the referenced memory location (multiplicand) and places the product in the QA Registers. The contents of the memory location is not changed.

Problem: Multiply the contents of MEM by the quantity 20_{16} which is currently in A. Store the results of A in memory location A and the results of Q at memory location Q.

Location	Opcode	Address	Comments
	MUI	MEM	
	STA	A	
	STQ	Q	

4.5 LDQ AND DVI INSTRUCTIONS

The Load Q (LDQ) instruction places the contents of the referenced memory location into the Q register. Contents of memory are unchanged.

The Divide Integer (DVI) instruction divides the 32-bit QA Register by the contents of the referenced memory location. The contents of memory are not changed. Q will contain the remainder and A the quotient.

Problem: Divide 4528_{16} by the contents of location SAVE

Solution:

Location	Opcode	Address	
SAVE	NUM	$0025	Divisor
TAG1	NUM	$4528	Dividend
TAG2	NUM	$0000	
	LDQ	TAG2	Clear Q
	LDA	TAG1	Get dividend
	DVI	SAVE	Answer is in A

4.6 JMP AND RTJ TO ILLUSTRATE A SUBROUTINE

The Jump (JMP) instruction causes a program sequence to terminate and initiates a new sequence at a specified location.

The Return Jump (RTJ) instruction is a jump enabling the program to begin execution in a new location and, by storing a return address, return to the next instruction in the program sequence.

Problem: During the main program sequence, skip to a routine to clear a storage area, and return to the location which the program left.

4.6

```
Location          Opcode       Address      Comments

                  LDA          NAME
                  STQ          SAVE
                  RTJ          CLEAR        Leaves P+2 in CLEAR.
                   |                        Program Control given
                   |                        to LDA TAGY.
                   |
                  SLS
CLEAR             0            0            Return address placed
                  LDA          TAGY         here.
                  STA          TAG2         Clear location(s)
                   |
                   |
                  JMP          (CLEAR)      Will return control to the
                   |                        SLS instruction.
                   |
                   |
TAGY              NUM          0
TAG2              BSS          TAG2(1)
```

5

CHAPTER V

1700 MACHINE INSTRUCTIONS

CHAPTER V - 1700 Machine Instructions

INTRODUCTION

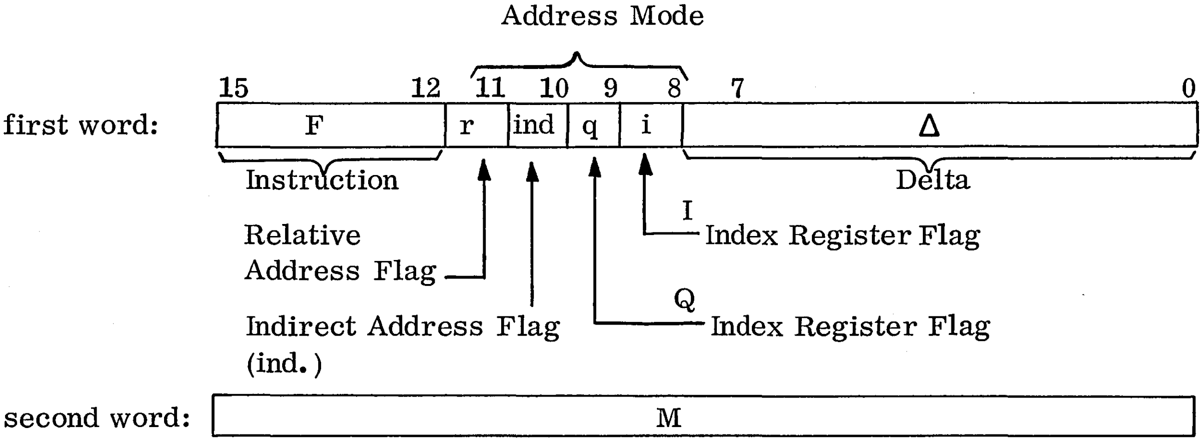
There are five classes of instructions in the 1700. They are:

Shift Class Instructions
Skip Class Instructions
Storage Reference Class
Register Reference Class Instructions
Interregister Class Instruction

Only the storage reference class uses core for operands. Instructions in all classes other than storage reference class are one word instructions and take 1.1 microseconds to execute with the exception of the shift class instructions whose execution time depends on the number of bits shifted, and INP and OUT instructions. Refer to Appendix B for instruction execution times, or to section 5.1.3.

5.1 STORAGE REFERENCE CLASS

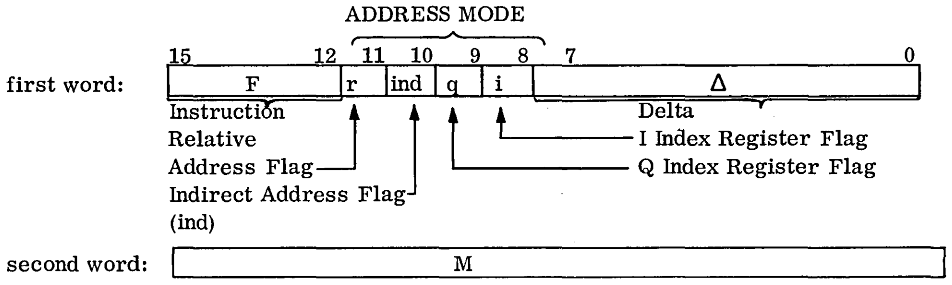
Of the five classes of instructions in the 1700, only this group uses core for operands. This means, then, that the instructions in this group are the only ones that are addressable. Because of the limitation of 16 bits in an instruction word to implement the ability to reach any core location from any other core location, two-word instructions are used. One-word instructions are available for addressing either absolute core blocks or within a fixed range of core locations from the instruction. The format for these instructions is:



Δ is not zero for one-word instructions.
Δ is zero, and M contains the operand (or operand address) for two-word instructions.

The instructions are defined by the F field and will be discussed later. The rest of the format word deals with various ways of addressing the instruction; this will be discussed first.

Figure 12. Address Modes for Storage Reference Class Instructions



Δ ≠ 0 for one-word instructions
Δ = 0 for two-word instructions

**	***	****
C000 0203	C000 2000	C000 4142

	Assembly Language Designator			Example		
Name	Opcode Terminator	Address Field	Operand Address	Hex Code	Instruction	Comments
Constant		=N	P+1	**	LDA =N$203	Numeric constant $203 → A
		=X		***	LDA =XBUF	Address of BUF → A
		=A		****	LDA =AAB	ASCII codes for AB → A
One-Word Absolute	-		Δ	C0FF	LDA- $FF	Contents of location $FF → A
One-Word Abs. Ind.	-	( )	(Δ)	C4FF	LDA- ($FF)	Loc $FF contains address of operand; operand is loaded into A
Two-Word Absolute	+		M	C400 0106	LDA+ $106	Contents of location $106 → A
Two-Word Abs. Ind.	+	( )	(M)	C400 8106	LDA+ ($106)	Loc $106 contains address of operand; operand is loaded into A
One-Word Relative	*		P+ Δ	C845*	LDA* BUF	Contents of loc BUF → A; BUF is within 127_{10} locs of the LDA instr.
One-Word Rel. Ind.	*	( )	(P+ Δ)	CC45	LDA* (BUFADR)	Loc BUFADR contains addr of operand; operand in BUF → A; BUFADR is within 127_{10} locs of LDA instr.
Two-Word Relative			P+1+M	C800 0044	LDA BUF	Contents of loc BUF → A; BUF is any distance from the LDA instr.
Two-Word Rel. Ind.		( )	(P+1+M)	CC00 0044	LDA (BUFADR)	Loc BUFADR contains addr of operand; operand → A; BUFADR is any distance from the LDA instr.

*If the buffer is 45_{16} locations in the positive direction from this instruction

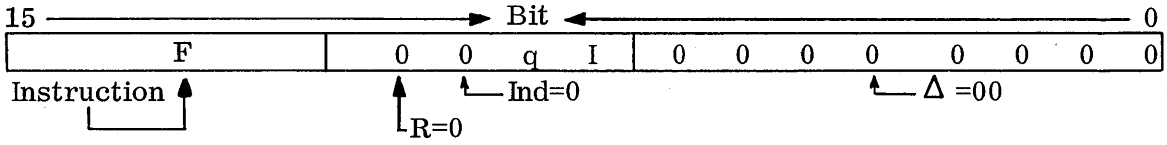
5.1.1 Addressing for Storage Reference Class Instructions

Figure 12 illustrates the various ways of addressing storage reference class instructions. The combination of the r, ind, and Δ bits determines the mode of addressing used. Because of the limitation of 16 bits in an instruction word, some modes of addressing require two words to reach all parts of core. A brief glance at this chart shows a direct correspondence between two-word instructions and the bit content of Δ. When Δ (bits 0-7) in the instruction word is all 0's, two consecutive core words are used as a single instruction. Also, there are only three main types of addressing: constant, absolute and relative. Within the absolute and relative types are one and two-word varieties and also one and two-word indirect types. Listed also in this table are the proper assembly language designators necessary to tell the assembler the exact mode of addressing desired. Indirect addressing is specified by parentheses enclosing the address field contents. This table ignores manipulations with the index register since index register modification is common to all modes of addressing. This figure gives only the base address for each mode. The base address must be found before index register modification occurs even when indirect addressing is specified. In each of the indirect addressing modes notice that the base address is specified as the contents (the use of parentheses around the address specifies the contents of) of its corresponding non-indirect mode. This means that for indirect addressing a further search is made into the addressed core location to find the base address.

5.1.1.1 Constant Mode

This mode of addressing is used where an operand is known to the programmer, that is, he is using a constant. This mode of addressing utilizes two words.

The first word of the instruction is specifically the instruction itself, the mode of addressing for this example is constant, and the unused bits in this word will be set to zero. This arrangement is illustrated below:



The upper four bits signify the actual instruction. This field is called the F field. The next four bits, that is, bits 8, 9, 10 and 11, are used to signify the mode of addressing for the instruction. Also, bits 8 and 9 will signify indexing, where bit 8 signifies index register I which is core location FF, and bit 9 will indicate indexing with the Q register. For the constant mode of addressing bit 10, which is called the indirect bit, is a zero and bit 11, which is called the relative bit, is also a zero. For our example here, we will assume no indexing used, therefore bits 8 and 9 will be 0. The lower eight bits, that is bits 0-7, are called the Δ field, and for constant mode of addressing all these bits are 0's.

The combination of the Δ field, the indirect bit and the relative bit are indicators to the machine of the mode of addressing used. Note for constant mode the field is all 0's, the indirect bit is a 0 and the relative bit is a 0, signifying constant mode of addressing.

Numeric constants. This form is for numbers.

Example:

```
LDA         =N$407F
```

This constant mode instruction written in assembly language will generate two machine words of code:

```
at P        C000
   P+1      407F
```

The operand itself is placed in the second word, so the base address is P+1, the second word of the instruction.

The =N in the address field signifies to the assembler the constant as numeric (a number). The number can be either decimal or hexadecimal (in which case it is preceded by the dollar sign). The result of the above example is to load into A the number $407F_{16}$.

Example:

```
LDA         =N256
```

The decimal number 256 would be converted and the following code generated:

```
at P        C000
   P+1      0100
```

Illegal example:

```
LDA         =N$100+$27
```

An error message would result because address arithmetic (+) is not allowed with the =N form of constant addressing.

Address constants. This form is for 15-bit addresses and address arithmetic on numbers.

Example:

```
LDA         =XTAG
```

The =X in the address field signifies that TAG is a symbol. The assembler substitutes the value in its symbol directory for the symbol and puts this value in P+1. If, for example, TAG is a location symbol at address 100 in the program, the following code is generated.

```
        at P        C000
           P+1      0100
```

An expression can also be used:

```
           LDA      =XTAG+10
```

This will put the address of TAG+10 in the code:

```
        at P        C000
           P+1      010A
```

The =X form of address constants is the correct way to allow numeric address arithmetic:

```
           ADD      =X$34F2-022+$1A
```

will generate code to add to the contents of the A register $34F6_{16}$.

```
        at P        ~~C000~~ 8000
           P+1      34F6
```

Even though the result of the expression in the address field is a constant, the =X form must be used rather than =N because an expression is not allowed with =N. LDA   =X$100+$27 would be legal and would generate:

```
        at P        C000
           P+1      0127
```

In address expressions only a 15-bit constant will be produced so note that

```
           LDA      =X-TAG
```

will generate

```
        at P        C000
           P+1      7EFF         (not FEFF)
```

also, with numbers:

```
           LDA      =X-11
```

will generate

```
        at P        C000
           P+1      7FF4
```

This feature of the assembler can well be utilized in sophisticated coding.

ASCII constants. This form is for alphabetic and numeric characters.

Example:

```
           LDA      =AXY
```

The =A means that the alphanumerics following (only 2 allowed) are to be converted to their ASCII 8-bit code. (See Appendix E.) The first alphanumeric (X in this case) ASCII code is placed in the high order 8 bits of P+1, and the second ASCII equivalent is placed in the lower 8 bits. The code will be:

```
at P        C000
   P+1      5859
```

A blank is a character in the ASCII form:

```
   LDA      =A  X
```

will generate the code:

```
at P        C000
   P+1      2058
```

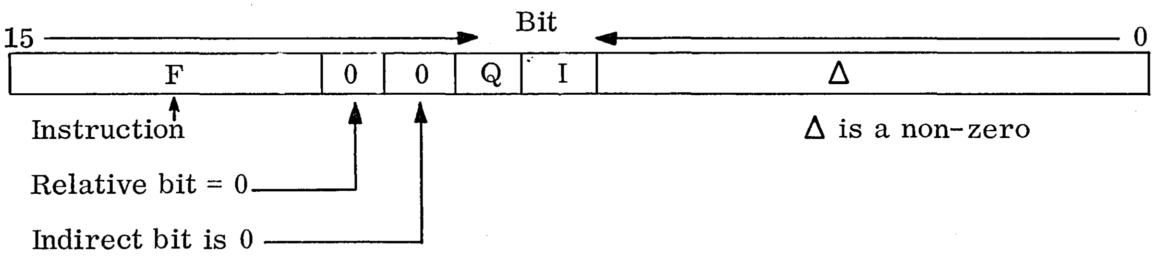
ASCII will be discussed in detail when I/O is discussed.

Examples:

C000 1000	LDA	=N$1000	Get 1000_{16} into A register
C000 00C8	LDA	=N200	Get 200_{10} into A register
C000 0500	LDA	=XDATA	Get address of DATA P0500 into A register
C000 0505	LDA	=XDATA+5	Get address of DATA+5 into A register
C000 7EFF	LDA	=X-$100	Get 15-bit negative of 100_{16} into A register
C000 5820	LDA	=AX	Get ASCII constant X into A register

5.1.1.2 Absolute: One Word

As was mentioned before, there are both one-word and two-word instructions for the Storage Reference Class. Let's examine the absolute mode of addressing in its one-word form. The format for this mode of addressing is:



The F field remains the same as in constant mode, that is, it signifies the type of instruction within this class used. The relative bit and the indirect bit are also 0's for this class. However, Δ is non-zero (See Figure 12). It may have been noticed now that this one-word type instruction has Δ non-zero where the constant mode example had Δ as zero and was a two-word type. The actual value in Δ will be the absolute address. Notice the size of Δ. It is only eight bits which means that the range of numbers in hexadecimal is from 01 to FF. In decimal this gives a range from 1 to 255. This is the limitation on this mode of addressing. Its advantage is its one-word length. To implement this absolute region from core location 01 to FF, all operating systems for the 1700 define this area as being the Communications Region. In this region are placed all system masks and all references to other points in the system allowing quick access through this region with this mode of addressing.

Example:

```
START               LDA-            $21
                    STA-            $EC
```

The contents of location 21_{16} in the absolute communications region is moved to core location EC_{16}. The minus sign as an opcode terminator signals the assembler to form a one-word instruction with the Δ field set to the address. The example above generates two one-word instructions:

```
                                    C021
                                    60EC
```

Example:

```
                    LDA             =N0
                    STA-            $FF
```

This example clears index register I. Since I is core location $FF which is in the communications region, all references to I will use absolute one-word addressing.

5.1.1.2

The base address, for one-word absolute mode of addressing cannot be 0 since a delta of 0 results in a two-word instruction at run time.

Example:

```
                LDA-            $4F-$50+1
```

This results in base address of 0 which is an assembly error. The base address must be 01 to FF at run time.

5.1.1.3 Absolute: One Word Indirect

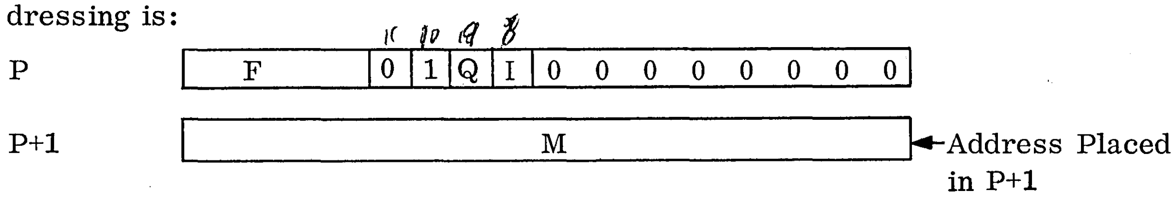
This mode of addressing is the indirect version of the absolute one-word. Its format, then, will be the same as for one-word absolute; however, bit 10 (the indirect bit) is set. Its base address, then, is not Δ but the contents of Δ and further, if the high order bit of the contents of Δ is a 1 then the indirect search will continue. This mode of addressing is extensively used where locations in the communications region contain addresses of other programs. The reference to these other programs is made by the use of the one-word absolute indirect mode of addressing through the communications region.

Example: Assume location E_6 in the communications region always contains the address of a desired routine. The routine can then be entered by:

```
                JMP-            ($E6)
```

5.1.1.4 Absolute: Two Word

Since it is necessary to be able to absolutely address any core location from any other core location, it takes a second word for the absolute mode of addressing to specify the absolute address. The format for absolute two-word mode of addressing is:



r = 0
ind = 1
$\Delta = 00_{16}$

Again, F will specify the particular instruction, Δ will be 0, the relative bit (bit 11) is 0, and the indirect bit (bit 10) is set. The second word of the instruction will contain the actual absolute address. Since the limit for allowable core in the 1700 is 32,767, only 15 of the 16 bits in the second word are utilized for this absolute address. For absolute two-word bit #15 of the second word will be 0.

Example:

```
COUNT              0                  0
START              LDA+               COUNT
```

The + opcode terminator signals the assembler that the base address must be placed in P+1. The base address, then, is (P+1). Notice (Figure 12) that this is really the indirect case for constant mode.

The address COUNT is placed in the second word. If the assembler's program counter for COUNT is at 0033, then at 0034 the assembler would generate:

```
                   C400
                   0033

START              STA+               $7F32-41
```

The address is calculated as $7F32_{16}-29_{16}$ or 7F09 and this address is placed in the second word of the instruction:

```
   At START location                  6400
                                      7F09
```

Example: Add 1000_{10} to the contents of core location 1000.

```
                   LDA+               $1000
                   ADD                =N1000
                   STA+               $1000
```

5.1.1.5 Absolute: Two Word Indirect

With indirect addressing the base address does not contain the operand itself, but rather this base address contains the address of the operand. Notice from Figure 12 that two-word absolute is actually a case of constant mode indirect. Instead of base address being P+1 as in constant mode, the search is made into P+1 for the base address. The indirect bit which is set for two-word absolute mode is actually a case of constant mode indirect. Indirect addressing, however, can be multi-level; that is the search may continue from address to address to find the final base address. The continuation of this indirect search is accomplished each time bit 15, the high order bit, of the base address is a 1 and the indirect bit (bit 10) is set. Using this high order bit of the address as an indirect flag is possible since only the low order 15 bits of this address can contain another address. Since the indirect bit is already set for the two-word absolute mode of addressing, the use of parentheses in the address field for this mode of addressing will cause bit 15 of the second word to be set. This forms two-word absolute indirect mode of addressing.

Example:

```
TAG                -                  -                  Assume TAG contains 0400
START              LDA+               (TAG)
```

If TAG is at the absolute location 0301, this code is generated:

302	C400
303	8301

The high order bit of 303 is set. The contents of 303 is examined at run time and since the high order bit is set, the search for a base address continues. The contents of 301 is brought out, and if the high order bit is a 1, the search would continue; in this case, however, the high order bit is a 0, so 400 is the base address. The effect of this instruction, then, is to load A with the contents of 0400.

Example: Assume the following values in core:

Core	Value
500	C400
501	C010
-	-
-	-
400F	3407
4010	8501

If the contents of core location 500 were executed as the first word of a two-word instruction, the computer would be in an endless loop searching for an address. The instruction at core location 500 is:

```
LDA+            ($4010)
```

Since this is two-word absolute indirect, the search continues to core location 4010 for an address. Bit 15 is set in core location 4010, so the search continues to 501, then back to 4010, etc. This condition is catastrophic, of course, but it illustrates the fact that the search for an address will continue until bit 15 is a 0; then this cell contains the base address.

5.1.1.6 Relative: One Word

There are two types of relocation associated with programs. One is called program relocation which means that the assembler begins the assembly with its program counter equal to zero so that this program, when it is loaded into core by the relocating linking loader, can be relocated anywhere. This program relocation is strictly a function of the assembler and the loader. This ability allows the program to be loaded anywhere into the core and run. Once loaded, the ability to take the same program and move it from one area of core to another and the program still run, is not a function of the loader. This type of relocation is known as dynamic relocation or "run anywhere". In process control programs, generally many programs are put on a mass storage device with a common area of core allotted for running these programs at any time. To allow proper allocation of this core area the programs on the mass storage device should be run anywhere so they can fit into space available in the common area rather than in a particular area of the common core. Achievement of run anywhere programs is the result of the use of the relative mode of addressing in the 1700. For the one-word relative mode of addressing Δ will contain a signed increment that when added to P will yield the base address. The base address is made relative to P or where the program instruction presently is so that if the whole program is moved, the same relative distance is maintained between instructions/data and the base address found in exactly the same manner, since P is variable. The format for the one-word relative commands is:

F	1 0 Q I	Δ

Contains signed 8-bit increment

The limitation for this one-word relative mode of addressing is the size of Δ. Since Δ is 8 bits and signed, the range from P is +7F to 80 (-7F), or $\underline{+}127_{10}$.

Example:

```
LOOP          -
              -
              -
              JMP*          LOOP
```

1011 1100

-2 = 0000 0010
= 1111 1101

If the JMP instruction is -127_{10} locations back or less, the one-word relative form can be used (and is preferred). Assume the program counter is at 0100_{16} for LOOP and at 0143_{16} for the JMP instruction, then the JMP instruction decodes at:

18BC   Note: Δ is BC which is -43_{16} in 8 bits.

The * opcode terminator is used to signify to the assembler one-word relative mode of addressing.

Example: Consider the same basic structure but coded absolutely, then relatively. Assume the distance from LOOP to LAST is less than 127_{10}.

```
a)     LOOP          -           -
                     -           -
                     -           -
       LAST          JMP+        LOOP

b)     LOOP          -           -
                     -           -
                     -           -
       LAST          JMP*        LOOP
```

For a) above, the routine could not be moved to another core location once loaded because with two-word absolute used with the JMP instruction, the absolute address placed in the second cell would cause a jump back to the original location of LOOP. But in case b), the incremental difference placed in Δ for JMP* LOOP, and since the distance between LOOP and LAST will not change if the whole routine is moved, the program will jump to the new location of LOOP. b), then, is "run anywhere" where a) is not.

5.1.1.7 Relative: One Word Indirect

This mode of addressing is an extension of the one-word relative to the indirect mode; the relative bit is set, the indirect bit is set, and Δ is non-zero (see Figure 12). With the base address found as in one-word relative ($P+\Delta$) a further search is made in this address location for the base address. The indirect bit is set and the search will continue if the high order bit in the contents of $P+\Delta$ contains a one.

Example:

```
ADDR        -          -          Contains an address-
                                  Assume 0600.
            -          -
            -          -
            -          -
            JMP*       (ADDR)
```

For this example ADDR must be within 127_{10} of the JMP instruction and ADDR contains the address, 0600, to which control will pass. This routine jumps to location 600 and continues program execution from there.

5.1.1.8 Relative: Two Word

Two-word relative mode of addressing is used when the difference between the instruction and the address is greater than the limitations imposed by one-word relative, that is greater than $\pm 127_{10}$. Two words are necessary and in the second word of the instruction is placed the difference between that word and the address. The base address is then P+1+M where M is the contents of P+1.

Example:

```
LOOP        - \ ─────────────────────► Assume PC-0100
            -  \
            -   >──────────────────── More than 127₁₀
            -  /
            - /
            JMP        LOOP ─────────► Assume PC-0400
```

Two word relative mode of addressing is used here since the difference between the JMP instruction and LOOP is greater than -7F. The above JMP LOOP would decode as:

```
400-       1800
401-       FCFE
```

Notice that the difference between P+1 and LOOP is placed in P+1.

5.1.1.9 Relative: Two Word Indirect

This mode of addressing is the extension of the two-word relative to the indirect mode; the relative bit is set, the indirect bit is set and Δ=00 since this is a two-word instruction. With the base address found as in two-word relative, a further search is made in this address location for the base address. The base address is then (P+1+M). The search will continue if the high order bit of the contents of P+1+M contains a one since the indirect bit is set.

Example:

```
ADDR        -          -          Contains an address-
                                  Assume 0600.
            -          -
            -          -
            JMP        (ADDR)
```

ADDR can be any distance from the JMP instruction and ADDR contains the address, 0600, to which control will pass.

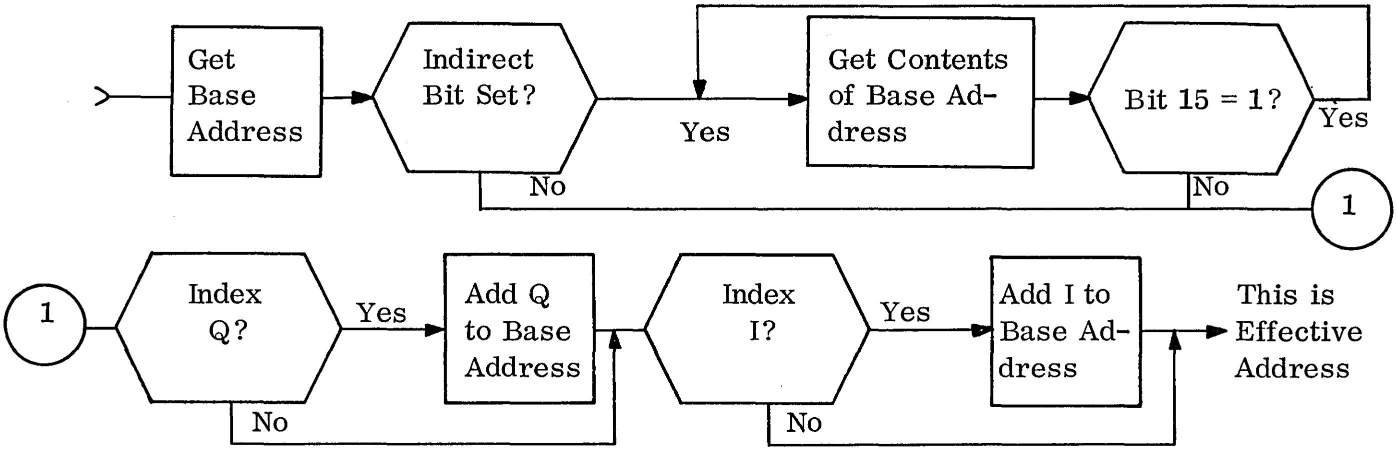
5.1.1.10 Review of Addressing Modes

A cross check back through Reference 1, Appendix A, page 3-4 will show that the Computer Reference Manual describes the storage reference class as having 7 modes of addressing. A count from Figure 12 here shows 9 distinct modes of addressing. Although a seeming conflict exists, there is really none. From the viewpoint of basic machine language, there are only 7 modes since two-word absolute indirect to the basic machine is nothing more than an extension of two-word absolute in the indirect mode and both relative modes can be grouped together as one. When considering the manner in which the programmer must specify the different modes to the assembler, there are 9 different combinations as described. Some of the terminology also differs between this training manual which essentially follows the terminology of the assembler manuals and the terminology used in the computer reference manual. The basic difference in terminology is:

Storage Mode is the same as Two-Word Absolute
Absolute Mode is the same as One-Word Absolute
Indirect is the same as One-Word Absolute Indirect
Relative Mode is the same as One-Word Relative
16-Bit Relative is the same as Two-Word Relative.

5.1.1.11 Indexing

Two index registers are available, the Q register and the I register which is actually core location FF. The contents of these registers can be used to modify the base address to form an effective address. This indexing, or address modification is accomplished simply by adding the contents of the specified index register (either Q or I or both) to the base address. If indirect addressing is specified, the search is made to find the final base address before indexing is done. This simplified flow chart illustrates this:



The use of indexing is wide-spread principally for forming repetitive loops for the type of work that would otherwise require an inordinate amount of repetitive programming.

Example:

Suppose the problem is to add together a series of numbers located in consecutive memory locations 1000 thru 1002 and then store the result into a location called TEMP. This routine would accomplish the additions:

```
TEMP        0        0              Reserve Temporary Lo-
START       LDA+     $1000          cation
            ADD+     $1001
            ADD+     $1002
            STA*     TEMP
```

If the series of numbers to be added was much longer, however, the length of the program, due to the number of ADD instructions, would be prohibitive. The same problem, with the numbers to be added in core 1000 thru 2000, ignoring overflow, and with the instructions covered so far would be solved with an indexed loop:

```
TEMP        0        0              Reserve temporary loc.
START       LDQ      =X$2000-$1000  Difference in Q
            LDA+     $1000
LOOP        ADD+     $1000,Q
            ADQ      =N-1           Decrease Q by 1
            SQZ      DONE-*-1       Finished?
            JMP*     LOOP           No
DONE        STA*     TEMP           Yes
```

Here, Q is set up as the index register used. The , Q in the address field signifies Q register indexing, and the assembler will set bit 9 in this instruction word. Q contains initially the difference between the lower and upper core locations of the numbers to be added. For each pass through the loop, Q is decreased by 1 allowing the next lower core location contents to be added in turn. The sequence of addition is: (1000) from the LDA instruction, (2000) from B. A.* of 1000 +Q which is initially 1000, (1FFF) from B. A. of 1000 +Q which is now 1 less or 0FFF, (1FFE)---, till (1001). When Q is decreased to 0, the SQZ instruction skips out of the loop.

If Q were not available at this point in the program, I could be used:

```
TEMP           0              0              Temporary storage
START          LDA            =X$2000-$1000
               STA-           $FF            Index Register I
               LDA+           $1000
LOOP           ADD+           $1000,I
               STA*           TEMP
               LDA-           I              Can use either I or $FF
               ADD            =N-1
               STA-           I
               SAZ            ENDIT-*-1
               LDA*           TEMP
               JMP*           LOOP
ENDIT                                        Result is in TEMP
               -              -
```

The , I in the address field signifies to the assembler that I register indexing is to be used and bit 8 in the machine language word is set.

I cannot be manipulated as easily as Q. TEMP is used to store the partial results of the adds when the A register is used to manipulate the contents of the index register I.

The , B in the address field signifies to the assembler that both Q and I indexing are to be used.

The use of Q as an index register does not increase the execution time, but use of I indexing takes 1.1 microseconds longer.

*Base address

5.1.1.11

Addressing Problem

Given are contents of the index registers and some core locations. Contents of any locations not shown is zero. What will the A register contain after each instruction is executed?

(I)=0020
(Q)=0120

(1000) = 0120
(0240) = 1234
(1234) = 02ED
(0260) = 1111
(1254) = 2311
(1111) = 9000

(A)=

a. LDA+ $240 ________________

b. LDA+ ($240) ________________

c. LDA+ ($240), I ________________

d. LDA+ ($1111), B ________________

```
*                                                                              *
*                                STORAGE REFERENCE                             *
*             F   RIQI   DELTA                             M                   *
*        (P)=**** **** **** ****          - (P+1)=**** **** **** ****          *
*            ↑↑↑↑ ↓↓   ↓↓↓↓ ↓↓↓↓→→                                             *
* →→→→→→→→→→→↑↑↑↑ ↓↓→→→→→→→→→→   ↓                                             *
* ↑ *MNE**OPERATION*         ↓   ↓  *MODE*                 *BA*    *OP**ADR*   *
* ↑                          ↓   ↓                                             *
* 1  JMP EFA→P               0  =00 CONSTANT*               P+1         =      *
* 2  MUI (EFA)X(A)→QA        0  ≠00 ABSOLUTE                DELTA    -         *
* 3  DVI (QA)/(EFA)→A,REM→Q  4  =00 STORAGE                 M        +         *
* 4  STQ (Q)→EFA             4  =00 STORAGE INDTRECT        (M)      +  ( )    *
* 5  RTJ P+1∨2→EFA,EFA+1→P   4  ≠00 INDIRECT                (DELTA)  -  ( )    *
* 6  STA (A)→EFA             8  =00 LONG RELATIVE           P+1+M              *
* 7  SPA (A)→EFA,PARITY→A    8  ≠00 RELATIVE                P+DELTA  *         *
* 8  ADD (A)+(EFA)→A         C  =00 LONG RELATIVE INDIRECT  (P+1+M)     ( )    *
* 9  SUB (A)-(EFA)→A         C  ≠00 RELATIVE INDIRECT       (P+DELTA)*  ( )    *
* A  AND (A)∧(EFA)→A         ↓   *EFA*                                         *
* B  EOR (A)⊻(EFA)→A         +0   BA                                           *
* C  LDA (EFA)→A             +1   BA+(I)                                       *
* D  RAO (EFA)+1→EFA         +2   BA+(Q)                                       *
* E  LDQ (EFA)→Q             +3   BA+(Q)+(I)                                   *
* F  ADQ (EFA)+(Q)→Q                                                           *
*                                                                              *
```

5.1.2 Instructions in the Storage Reference Class

The instructions for the storage reference class are:

F			F		
1	JMP	Jump	9	SUB	Subtract from A
2	MUI	Multiply	A	AND	AND with A
3	DVI	Divide Integer	B	EOR	Exclusive OR with A
4	STQ	Store Q			
5	RTJ	Return Jump	C	LDA	Load A
6	STA	Store A	D	RAO	Replace Add 1 in Storage
7	SPA	Store A, Parity to A			
			E	LDQ	Load Q
8	ADD	Add to A	F	ADQ	Add to Q

Any of the nine addressing modes, plus indexing, can be used on all of these instructions.

5.1.2.1 LDA, LDQ, STA, STQ, AND SPA Instructions

These instructions are the basic load and store instructions. For the load instructions, either LDA or LDQ, the contents of the effective address specified is loaded into the registers, either A or Q, as per the specific instruction and the contents of the storage location are not altered. The STA and STQ registers store the contents of the register into the effective address, replacing what was in this core address with the information that was in the register. For the store instructions STA and STQ, the original contents of the A or Q registers are not changed. The SPA instruction is the same as the STA instruction with the exception that after the information from A is stored into the core address specified, A is cleared and the parity of the original contents of A is returned to bit 0 of A. This parity bit is the value that would be necessary to make the total number of bits of the original contents of A plus this bit, odd. This parity bit is not necessarily the same as the parity bit that will appear in core for this data word since core might contain a program protect bit. In fact, this bit is the exclusive OR of the core parity bit and program protect bit.

Any method of addressing may be used with these instructions.

Example:

Location	Opcode	Address	Comments
	LDA	TAG	

Contents of A is replaced by the contents of core location labeled TAG. The contents of TAG is not changed.

Location	Opcode	Address	Comments
	SPA	TAG1	

The contents of A (assume F2F7) is stored into the core location labeled TAG1. Then A is cleared and the parity of the original contents of A is returned to A, bit position 0. The parity returned is 1.

5.1.2.2 ADD, ADQ, SUB, MUI and DVI Instructions

These instructions are the primary arithmetic instructions for the 1700. Data from core can be added to the A register with the ADD instruction, or to the Q register with the ADQ instruction. Data from core can be subtracted from the A register with the SUB instruction. Notice there is no instruction for direct subtraction from Q. These arithmetic operations can cause overflow (refer to Chapter II, Section 4).

Example:

Location	Opcode	Address	Comment
	LDQ	TAG	
	ADQ	=N-6	

Using Constant mode of addressing for the ADQ produced an effective subtraction from Q. Q is loaded with the contents of core location TAG and the number -6 is added to Q.

The MUI instruction will multiply an operand in core by the contents of the A register. Both of these operands are 16 bits long, producing a double length, 32 bit product in the Q and A registers. The Q register contains the most significant portion of the product and the sign of the product and the A register contains the least significant portion of the product. The MUI instruction cannot cause overflow. The MUI instruction destroys the original contents of Q.

Example:

Location	Opcode	Address	Comment
	LDA	=N$FF31	-$CE to A register
	MUI	=N$31	Multiply by 31

Result is (Q register)=FFFF
(A register)=D891

The result in the double length register then is $-276E_{16}$.

Negative zero can be produced by the MUI instruction:

(+0) x (-N) = (-0)
(-N) x (+0) = (-0)
(-0) x (+N) = (-0)
(+N) x (-0) = (-0)

5.1.2.2

Division of integers is accomplished with the DVI instruction. It divides the double length QA register by the contents of a core location. The Q register must contain the most significant portion and the sign of the dividend and A must contain the least significant portion of the dividend. The signed quotient will be placed in the A register and the signed remainder will be placed in the Q register after division occurs. The DVI instruction can cause overflow, if the quotient is larger than 7FFF in absolute value.

Example:

Location	Opcode	Address	Comment
	LDQ	=N$FFFF	=N in address field indicates CONSTANT MODE OF ADDRESSING used. The number following =N is the operand.
	LDA	=N$FFF0	
	DVI	=N$FFFA	

This example divides -15 by -5. Notice that Q had to be set to extend the sign to 32 bits. Result is 0003 in A and 0000 in Q.

5.1.2.3 AND and EOR Instructions

These instructions perform logical manipulation of data from the specified core location with the contents of the A register. The AND instruction performs a logical product operation; its bit by bit truth table is:

A	B	A∧B
1	1	1
1	0	0
0	1	0
0	0	0

Example:

	15 Bit 0
Operand 1 --	1111 0010 1001 0111
Operand 2 --	0011 1111 1100 0000
Logical Product	0011 0010 1000 0000

The AND instruction is used to extract a certain field of bits. In the example above, a field from bit 6 through 13 is extracted from operand 1. Operand 2 contains ones in these bit positions and 0's in the remaining bit positions to block out the unwanted bits. Operand 2 is referred to as a mask.

Example: Examine bits 4 thru 7 of core location TAG for all zeroes.

```
Location      Opcode      Address       Comment

              LDA         TAG           Load (TAG) in A register
              AND         =N$00F0       AND with A Mask
              SAZ         YES-*-1       Test for zeroes
              JMP         NO
YES           ---         ---
              ---         ---
```

The EOR logical instruction performs an exclusive OR bit by bit. The logical rules for an exclusive OR are:

A	B	A⊻B
1	1	0
1	0	1
0	1	1
0	0	0

A 0 is produced as a result of a match between the two bits and a 1 is produced as a result of a mismatch. This instruction is frequently used to test for a particular bit pattern.

Example: Test for 10110 in bit positions 3 thru 7 of contents of core location DATA.

```
Location      Opcode      Address       Comments

              LDA         DATA
              AND         =N$00F8       Mask out all except 3-7.
              EOR         =N$00B0       Test for bits 10110
              SAZ         YES-*-1
              -
YES
```

If the exact bit pattern 10110 was present in bit positions 3 through 7, all the bits in both operands would match, and a positive zero will be in the A register and the program will skip to YES.

An instruction that performs an inclusive OR is not specifically available in the storage reference class. The truth table for the inclusive OR is:

A	B	AVB
1	1	1
1	0	1
0	1	1
0	0	0

A 1 is produced if a 1 appears in either or both operands. Notice that the inclusive OR logical function can be obtained by the combination of the logical product and exclusive OR functions.

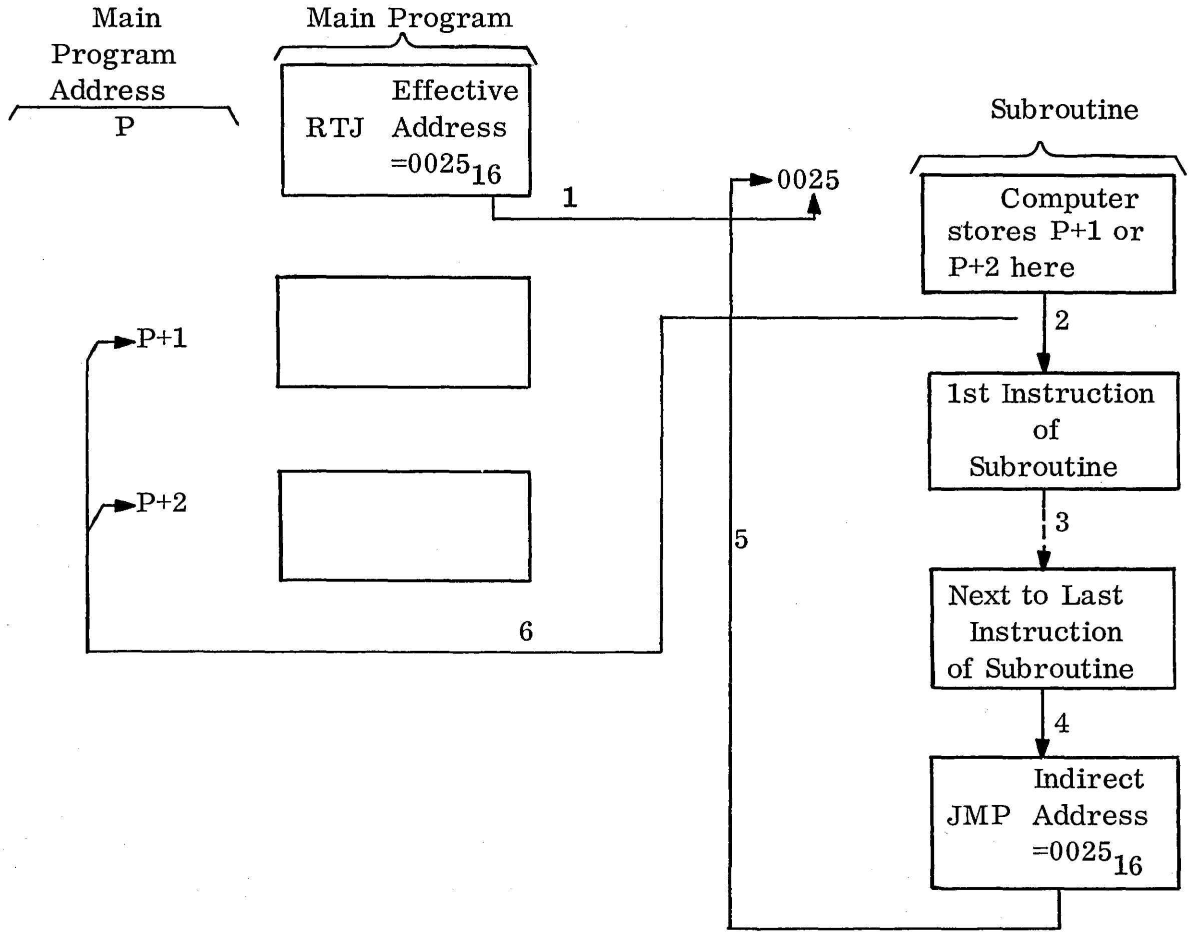
5.1.2.3

Example: Form the inclusive OR bit-by-bit of the contents of core location AA and BB.

Location	Opcode	Address	Comment
TEMP	0	0	
	LDA	AA	
	AND	BB	Logical Product AA & BB
	STA	TEMP	Store Temporarily
	LDA	AA	Exclusive OR AA & BB
	EOR	BB	
	ADD	TEMP	Combine them. Inclusive OR is now in A.

5.1.2.4 JMP and RTJ Instructions

These instructions alter the path of program flow. JMP is an unconditional program branch to the effective address. The RTJ, besides branching control to another core location, also provides the link by which control can be returned to the instruction following the RTJ instruction. This allows program flow from a main program to a closed subroutine and back again to the main program. This is illustrated below:



The subroutine is structured with its first location left open to allow the address (P+1 or P+2) to be placed by the RTJ instruction. Program control is then given to the second cell of the subroutine. By using a jump indirect through the first cell as the last instruction in the subroutine, program control will be given to the next executable instruction following the RTJ in the main program. This flow is independent of the location of the RTJ instruction.

Example:

Location	Opcode	Address	Comment
LOOP	LDA	TEN	
	-	-	
	-	-	
	JMP	LOOP	Simply jumps back to location LOOP
	-	-	
BACK	RTJ -	BACK1	Sets BACK+2 into BACK1 since mode of addressing is 2 word.
	-	-	
BACK1	0	0	Address BACK+2 is placed here.
	-	-	
	-	-	
	JMP	(BACK1)	Will jump not to location BACK1, but to location whose address is in BACK1 which is BACK+2.

To specify indirect mode of addressing the address is placed in parentheses as in the above example. This means that the specified address (in this case, BACK1) itself contains the address that is desired. More will be said on addressing later.

5.1.2.5 RAO Instructions

This RAO instruction is used to increase the contents of the effective address by 1. This instruction is very useful as a counter where a memory location is initialized or preset and the RAO instruction increases the count in this memory cell each time a desired condition occurs (number of times through a loop, etc.).

Example:

```
Location      Opcode       Address       Comment

COUNT         0            0
              -            -
              -            -
LOOP          LDA          TEST,Q
              -            -
              -            -
              SAZ          2             Skips to RAO COUNT if A
                                         is zero.
              JMP          LOOP
              RAO          COUNT
              JMP          LOOP
```

In this example the contents of core location COUNT is increased by 1 each time the A register is found to be zero. This RAO (Replace and Add One) instruction can only add one to the contents of the designated core location and does not change the contents of any of the registers. This instruction can cause overflow and, as such, is very useful for loop control.

Example: Loop through a given portion of a program 12 times then jump to core location NEXT. It is necessary to preset a COUNT cell such that increasing it by 1 will not cause overflow until it has been increased 12 times. The largest positive number minus 11 then is preset into COUNT.

```
Location      Opcode       Address       Comment

              SOV          0             Turn off overflow if set
              LDA          =X$7FFF-11
              STA          COUNT
LOOP          -
              -
              -
              RAO          COUNT
              SOV          NEXT-*-1
              JMP          LOOP
COUNT         0            0
NEXT          -
```

Overflow will occur the 12th time the RAO instruction is performed.

5.1.3 Execution Times

	INSTRUCTION	EXECUTION TIME (microseconds)*
LDA	Load A	2.2
STA	Store A	2.2
LDQ	Load Q	2.2
STQ	Store Q	2.2
ADD	Add A	2.2
SUB	Subtract	2.2
ADQ	Add Q	2.2
AND	AND with A	2.2
EOR	Exclusive OR with A	2.2
RAO	Replace Add One in Storage	3.3
MUI	Multiply Integer	7.0
JMP	Jump	1.1
RTJ	Return Jump	2.2
DVI	Divide Integer	9.0
SPA	Store A, Parity to A	2.2

Timings are for one-word instructions. An additional cycle must be added for a two-word instruction.

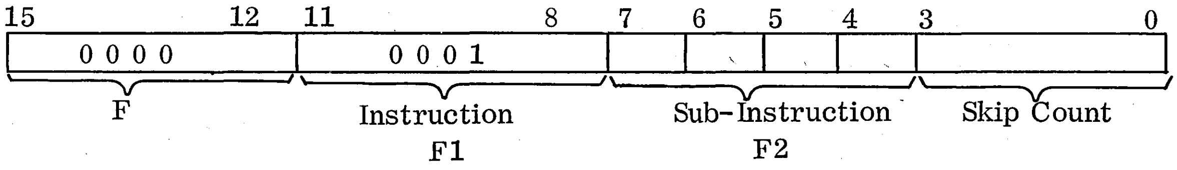
Note the speed of the integer multiply and divide instructions. These are considered very fast for the computer hardware.

*Add 1.1 microsecond if Storage Index Register is used.
Add 1.1 microsecond for each level of Indirect Addressing.

```
*                                   SKIPS                                          *
*                          F=0    F1=1   F2   SKIP  COUNT                          *
*                      (P)=****   ****  **** ****                                  *
*                                                                                  *
*  ↑    ARITHMETIC TESTS                     ↓         MACHINE STATE TESTS         *
*  ↑         *TEST*                          ↓          *TEST*                     *
*  0    SAZ    (A)=ZERO                      8    SWS    SKIP SWITCH ON            *
*  1    SAN    (A)≠ZERO                      9    SWN    SKIP SWITCH OFF           *
*  2    SAP    (A)=POSITIVE                  A    SOV    OVERFLOW                  *
*  3    SAM    (A)=NEGATIVE                  B    SNO    NOT OVERFLOW              *
*  4    SQZ    (Q)=ZERO                      C    SPE    PARITY ERROR              *
*  5    SQN    (Q)≠ZERO                      D    SNP    NOT PARITY ERROR          *
*  6    SQP    (Q)=POSITIVE                  E    SPF    PROTECT FAULT             *
*  7    SQM    (Q)=NEGATIVE                  F    SNF    NOT PROTECT FAULT         *
*                                                                                  *
```

5.2 SKIP CLASS INSTRUCTIONS

These instructions are used to make conditional tests and skip forward depending on whether the instruction meets the actual condition being tested. The format for these instructions appears below:



Notice the skip count is only four bits which allows a skip count of only 15. The skip count is not signed so the instructions will only allow a skip in the forward direction.

If the skip condition is met, program control proceeds to P + 1 + skip count. If the skip condition is not met, program control continues from P+1.

5.2.1 A and Q Skip Tests

The following types of conditional tests are available on the A and Q register:

F2			
0	0000	SAZ	A=+0
1	0001	SAN	A≠+0
2	0010	SAP	A= +
3	0011	SAM	A= -
4	0100	SQZ	Q=+0
5	0101	SQN	Q≠+0
6	0110	SQP	Q= +
7	0111	SQM	Q= -

Example:

Location	Opcode	Address	Comments
START	LDA	=N$0080	Loads 0080 into A
	ARS	4	
	SAZ	1	1 is the Skip Count
	ARS	4	will not skip to TAG
TAG	SAZ	TAG1-*-1	will skip to TAG1
	-		
	-		
	-		
TAG1			

The TAG1-*-1 address is a form peculiar only to the utility assembler. With this form the assembler is being directed to form a skip count to TAG1, which has some program counter value. This counter value, minus the current location of the SAZ instructions (which is some program counter value less than TAG1), minus one more, forms the skip count. The -1 is used to compensate for the +1 in the skip formula P+1+skip count. When using the macro assembler, the address need only be TAG1 as the macro assembler will automatically calculate the skip count. For all cases the distance skipped (to TAG1 in this example) must be no greater than 16 locations forward from the location of the skip command. The macro assembler will also accept the -*-1 form.

5.2.2 Skip Switch Tests

The SWS and SWN instructions test the condition of the skip switch located on the programmer's panel. This switch can be used to alter program flow from the panel.

Example:

```
Location        Opcode          Address         Comments
START           -
                -
                -
TAG             SWS             GO-*-1
                -
                -
                -
GO              -
TAG1            SWN             GO1-*-1
                -
                -
                -
GO1             -
```

The code between TAG and GO will be skipped at run time if the skip switch on the programmer's panel is set. It would be executed if the switch were not set.

The code between TAG1 and GO1 will be skipped if the skip switch on the programmer's panel is not set. It will not be skipped if the switch is set.

5.2.3 Overflow Skip Tests

The SOV (F2=A) and SNO (F2=B) instructions test the state of the overflow indicator. Refer to Chapter II, Section 4, for a discussion of those arithmetic operations causing overflow. The overflow indicator is cleared upon execution of these instructions.

Example:

```
Location       Opcode        Address        Comments

START          LDA           OP1
               ADD           OP2
               SNO           TAG-*-1
CORR           -             -
               -             -
TAG            ADD           OP3
               SOV           ERROR-*-1
               JMP           OK
               -             -
ERROR          -             -
               -             -
               -             -
               -             -
OP1            -             -
OP2            -             -
OP3            -             -
OK             -             -
```

In the above example, two numbers are added and if no overflow occurred, the instructions between CORR and TAG will be skipped and a third number will be added. The overflow indicator is again checked and if overflow had occurred, the routine would skip to ERROR. The overflow indicator once set, remains set until tested with either an SOV or SNO instruction.

5.2.4 Parity and Program Protect Indicator Tests

The SPE (F2=C) and SNP (F2=D) instructions will test for a storage parity error and the SPF (F2=E) and SNF (F2=F) instructions will test for program protect fault errors.

Example:

```
Location       Opcode        Address        Comments

START          SPE           PAR-*-1
               SPF           PROT-*-1
               JMP           ERROR
PAR            JMP           PARITY
PROT           JMP           PROTEK
```

If either a parity error or a program protect fault occurs, an interrupt is generated on line 0. Since both interrupts can occur on the same line, the interrupt processor for this line must distinguish between the two. This example illustrates how this might be done with program control going to the location named PARITY if a parity error is found, and program control jumping to core location named PROTEK if the protection fault caused the interrupt. If neither interrupt was found by this process, program control would jump to some error routine to service what is apparently a ghost interrupt. The parity and protect indicators (both the interrupt signal and the programmer's panel fault indicators) will clear when these instructions are executed.

Problem: The following routine will move how many numbers? From what core locations to what core locations?

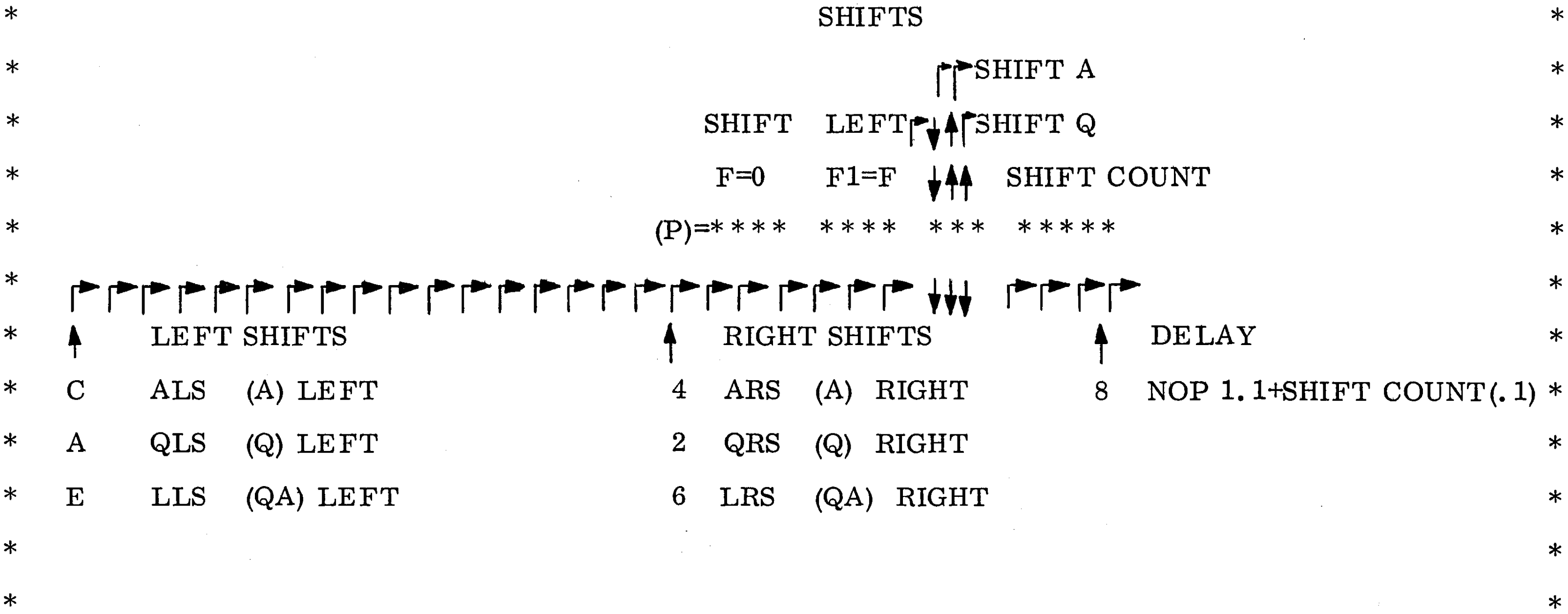
```
MOVE          0          0
              LDQ        =N$1000
LP1           LDA+       $1000,Q
              STA+       $3000,Q
              ADQ        =N-1
              SQM        1
              JMP*       LP1
              JMP*       (MOVE)
```

Problem: The following routine sums how many numbers? From what core locations? Where does it store the answer?

```
SUM           LDQ        =X$2000-$1000
              LDA        =N$
LOOP          ADD+       $1000,Q
              ADQ        =N-1
              SQM        DONE-*-1
              JMP*       LOOP
DONE          STA+       $3000
```

Problem: The following routine moves how many numbers? From which core locations to where?

```
CHNG          0          0
              LDA        =N$1000
              STA*       ADDR1
              LDA        =N$4000
              STA*       ADDR
LP2           LDA+       (ADDR1)
              LDQ+       (ADDR)
              STA+       (ADDR)
              STQ+       (ADDR1)
              RAO*       ADDR1
              LDA*       ADDR
              SUB        =N1
              STA*       ADDR
              SUB        =N$3000
              SAM        3
              JMP*       LP2
ADDR1         0          0
ADDR          0          0
              JMP*       (CHNG)
```



SHIFTS

SHIFT A

SHIFT LEFT SHIFT Q

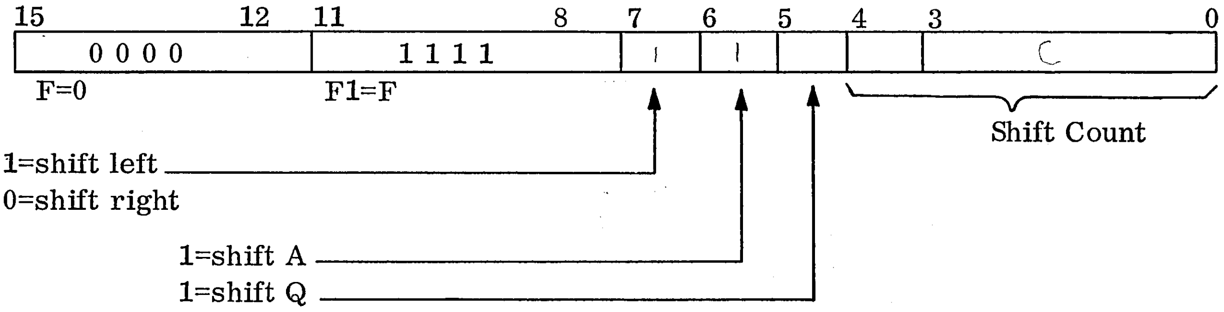
F=0 F1=F SHIFT COUNT

(P)=**** **** *** *****

LEFT SHIFTS			RIGHT SHIFTS			DELAY
C	ALS	(A) LEFT	4	ARS	(A) RIGHT	8 NOP 1.1+SHIFT COUNT(.1)
A	QLS	(Q) LEFT	2	QRS	(Q) RIGHT	
E	LLS	(QA) LEFT	6	LRS	(QA) RIGHT	

5.3 SHIFT CLASS INSTRUCTIONS

These instructions are used to shift the data bit by bit either in the A register or the Q register singly or together. The data can be shifted either left or right bit by bit. For these instructions the format appears below:



Notice the shift count is five bits allowing a shift either way a maximum of 31 positions. The upper bit of the shift count will appear in the hex code as part of the second digit. For example, a 0F51 is an ARS ~~Ø~~ 17 instruction, not ARS 1.

Left shifts are end around: the high order bit of the register is shifted around and into the low order bit of that register for single register shifts. For a double register shift, (Long Shift), the Q register is considered the most significant register and the A register the least significant register and on left shifts the high order bit of Q is shifted around into the low order bit of A. High bit of A is shifted left into the low order bit of Q.

Right shifts are end off with sign extension. Bits shifted off the right end are lost, and the sign bit of the register is extended from the left. For long right shifts, the low bit (Bit 0) of Q is shifted into the high bit (Bit 15) of A and the sign (Bit 15) of Q is extended from the left.

The mnemonics for the instructions in this class are:

ARS	A Right Shift
QRS	Q Right Shift
LRS	Long Right Shift (QA)
ALS	A Left Shift
QLS	Q Left Shift
LLS	Long Left Shift (QA)

Example:

Location	Opcode	Address	Comments
	ALS	8	

If the A register contained F302, execution of this instruction would shift the A register left 8 bits leaving 02F3. The high order bits of A moved end around into the low order bits of A.

	LRS	8	

If Q=8000 and A=AOF0, execution of this instruction extends the sign of Q to the right and the lower 8 bits of A would shift end off and be lost. Result is FF80 in Q, and 00A0 in A.

Illegal	LLS	40	Maximum number of shifts allowed is 31,
Legal	QRS	0	Is effectively a no operation

5.3.1 Timing for Shift Class Instructions

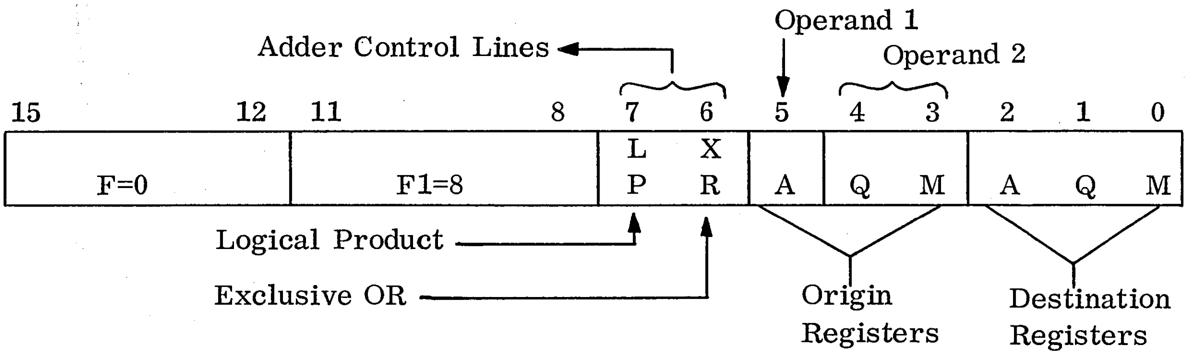
The time for shift class instruction execution is:

For long shifts (QA together) 1.1 + .2 x shift count
For single register shifts (Q or A) 1.1 + .1 x shift count

```
*                                INTERREGISTER                                        *
*                                          LX                                         *
*                                F=0   F1=8   PR   AQM   AQM                          *
*                            (P)=****  ****   **   ***   ***                          *
*                       ADDER CONTROLS→→|     ↓↓↓   ↓↓↓ →→DESTINATION REGISTERS        *
*                                             ↓↓↓ →→ORIGIN REGISTERS                  *
*                                             ↓↓↓ →→OPERAND TWO                       *
*                                             ↓→→OPERAND ONE                          *
*                                                                                     *
* TRANSFER                  TRANSFER COMPLIMENT            SUM                        *
* 40   CLR   0→A, Q, M      80  SET  -0→A, Q, M            30  AAQ  (A)+(Q)→A, Q, M   *
* A0   TRA   (A)→A, Q, M    60  TCA  (A)NOT→A, Q, M        28  AAM  (A)+(M)→A, Q, M   *
* 90   TRQ   (Q)→A, Q, M    50  TCQ  (Q)NOT→A, Q, M        38  AAB  B+(A)→A, Q, M     *
* 88   TRM   (M)→A, Q, M    48  TCM  (M)NOT→A, Q, M                                   *
* 98   TRB   B→A, Q, M      58  TCB  B  NOT→A, Q, M                                   *
*                                                                                     *
* EXCLUSIVE OR              LOGICAL AND                    COMPLIMENT LOGICAL AND     *
* 70   EAQ   (A)⊻(Q)→A, Q, M   B0  LAQ  (A)∧(Q)→A, Q, M    F0  CAQ  ((A)∧(Q))NOT→A, Q, M
* 68   EAM   (A)⊻(M)→A, Q, M   A8  LAM  (A)∧(M)→A, Q, M    E8  CAM  ((A)∧(M))NOT→A, Q, M
* 78   EAB   B⊻(A)→A, Q, M     B8  LAB  B∧(A)→A, Q, M      F8  CAB  (B∧(A))NOT→A, Q, M *
*                                                                                     *
*           B=INCLUSIVE OR OF (M)AND(Q).                                              *
```

5.4 INTERREGISTER CLASS INSTRUCTIONS

This class of instructions performs arithmetic or logical manipulation with the contents of A, Q or M or any combination of the three. The format for this class of instruction is:



Since the adder can operate on only two operands and there are three possible origin registers, these three origin registers are considered as two operands, operand 1 and operand 2. Operand 1 includes bit 5 or the A register bit and it can have two forms:

A (Bits)	-	Operand 1
0	-	FFFF
1	-	Contents of A

If this bit 5 is a 0, then all 1's are used as an operand and if bit 5 is a 1, then the contents of A is an operand.

Operand 2 is the combination of bits 3 and 4 or the combination of the Q and M register bits:

Q (Bit 4)	M (Bit 3)		Operand 2
0	0	-	FFFF
0	1	-	(M)
1	0	-	(Q)
1	1	-	Inclusive OR of Q & M

If neither of these registers is specified, then all 1's are used as the operand. If any one but not the other is specified, then the contents of that specified register is used. If both Q and M are specified, then the inclusive OR of Q and M is used as Operand 2. The bit by bit truth table for the inclusive OR is:

A	B	A V B
1	1	1
1	0	1
0	1	1
0	0	0

Here a bit in either position yields a bit in the result.

Either A, Q or M, or any combination of these can be specified as the destination registers. These are listed in any order and separated by commas in the address field.

Since M is the interrupt mask register, the interregister instruction with M as a destination register must itself be protected if the protect switch on the programmer's panel is on. Otherwise, a protect fault will occur.

Formation of the operation itself comes from bits 6 and 7 of the instruction word. The operations possible are:

LP (Bit 7)	XR (Bit 6)	Operation
0	0	Arithmetic Sum
0	1	Exclusive OR
1	0	Logical Product
1	1	Complement Logical Product

Refer back to Section 5.1.2.3 of this chapter for the truth tables of the logical product and the exclusive OR. All the possible combinations of different instructions in this class using these two operands number 22. The mnemonics assigned with the instructions in this class are:

SET	Set to 1's
CLR	Clear to "0"
TRA	Transfer A
TRM	Transfer M
TRQ	Transfer Q
TRB	Transfer $Q \vee M$
TCA	Transfer Complement A
TCM	Transfer Complement M
TCQ	Transfer Complement Q
TCB	Transfer Complement $Q \vee M$
AAM	Transfer Arithmetic Sum A, M
AAQ	Transfer Arithmetic Sum A, Q
AAB	Transfer Arithmetic Sum A, $Q \vee M$
EAM	Transfer Exclusive OR A, M
EAQ	Transfer Exclusive OR A, Q
EAB	Transfer Exclusive OR A, $Q \vee M$
LAM	Transfer Logical Product A, M
LAQ	Transfer Logical Product A, Q
LAB	Transfer Logical Product A, $Q \vee M$
CAM	Transfer Complement Logical Product A, M
CAQ	Transfer Complement Logical Product A, Q
CAB	Transfer Complement Logical Product A, $Q \vee M$

Examples:

```
LDA      =NO        Clears A
CLR      A          So does this
CLR      A,Q,M      Clears A, Q and M
AAQ      A          Adds A to Q, puts result in A
TCA      Q          Puts complement of A into Q
TRA      Q,M        Transfers A to Q and M
SET      M          Set M to all 1's
AAQ      0          Only affects overflow indicator--
                    adds A and Q, puts result nowhere
```

Problem:

The following is an example of how a subroutine can pick up parameters from the calling routine. How does the subroutine pick up the parameters? Does it pick up the actual argument or the address of the argument?

Calling Program:

```
                             LDA     =XX
                             STA     ARG1
                             LDA     =XY
                             STA     ARG2
                             LDA     =XZ
                             STA     ARG3

$500                         RTJ     SUB

$502       ARG1              0       0 700       Address of X

$503       ARG2              0       0 701       Address of Y

$504       ARG3              0       0 702       Address of Z

$700       X                 NUM     10          X data

$701       Y                 NUM     12          Y data

$702       Z                 NUM     6           Z data
```

Addresses of parameters fall directly beneath the call to the subroutine.

Subroutine:

```
SUB                 0                   0
                    STA*                SAVEA+1
                    STQ*                SAVEQ+1
                    LDA-                $FF
                    STA*                SAVEI+1
                    LDA*                SUB
                    EOR                 =N$8000
                    STA*                SUB
                    LDA*                (SUB)
                    STA*                SUBAG1
                    RAO*                SUB
                    LDA*                (SUB)
                    STA*                SUBAG2
                    RAO*                SUB
                    LDA*                (SUB)
                    STA*                SUBAG3
                    LDA*                SUB
                    INA                 I
                    AND                 =N$7FFF
                    STA*                SUB
                    ⋮
SAVEI               LDA                 =N0
                    STA-                $FF
SAVEQ               LDQ                 =N0
SAVEA               LDA                 =N0
                    JMP*                (SUB)
SUBAG1              BSS                 SUBAG1(1)
SUBAG2              BSS                 SUBAG2(1)
SUBAG3              BSS                 SUBAG3(1)
                    ⋮
```

Problem:

How many numbers does the following routine sort? In what order? From what core locations?

```
          SORT        CLR         A
                      STA-        $FF
                      ENQ         1
          BEGIN       LDA+        $500,Q
                      SUB+        $500,I
                      SAP         CHECK-*-1
                      LDA+        $500,Q
                      STA*        TEMP+1
                      LDA+        $500,I
                      STA+        $500,Q
          TEMP        LDA         =NO
                      STA+        $500,I
          CHECK       INQ         1
                      TRQ         A
                      EOR         =N$10
                      SAZ         1
                      JMP*        BEGIN
                      LDA-        $FF
                      INA         1
                      STA-        $FF
                      EOR         =N$F
                      SAZ         EXIT-*-1
                      ENQ         1
                      ADQ-        $FF
                      JMP*        BEGIN
          EXIT        SLS
```

```
*                                REGISTER REFERENCE                                      *
*                                 F=0    F1     DELTA                                    *
*                             (P)=****  ****  ****  ****                                 *
*  ↱↱↱↱↱↱↱↱↱↱↱↱↱↱↱↓↓↓↓ ↱↱↱↱↱↱↱↱                                                          *
*  ↑   ARITHMETIC             ↑  INTERRUPT         ↓  PROTECT           ↓       I/O      *
*  ↑                          ↑                    ↓                    ↓                *
*  A   ENA  +-DELTA   A       4  EIN*  ENABLE      7  CPB  CLEAR        2  INP  I/O↱A    *
*  C   ENQ  +-DELTA   Q       5  IIN   INHIBIT     6  SPB  SET          3  OUT  (A)↱I/O  *
*  9   INA  +-DELTA+(A)↱A     E  EXI*  EXIT        0  SLS  SELECTIVE STOP                *
*  D   INQ  +-DELTA+(Q)↱Q                          B  NOP                                *
*                              *ONE INSTRUCTION DELAY                                    *
*                                                                                        *
```

5.5 REGISTER REFERENCE CLASS INSTRUCTIONS

All the instructions in the class are one-word. The F field is always a zero and the F1 field will signify the particular instruction within this class. The format for the Register Reference Class of instructions is:

F=0	F1	Δ

Instruction Code

The instructions within this class are:

F1		
0	SLS	Selective Stop
1		SKIPS
2	INP	Input to A
3	OUT	Output from A
4	EIN	Enable Interrupt
5	IIN	Inhibit Interrupt
6	SPB	Set Program Protect
7	CPB	Clear Program Protect
8		INTERREGISTER
9	INA	Increase A
A	ENA	Enter A
B	NOP	No operation
C	ENQ	Enter Q
D	INQ	Increase Q
E	EXI	Exit Interrupt State
F		SHIFTS

The Δ field is available in this class of instructions.

5.5.1 Instructions ENA, INA, ENQ and INQ

These four instructions are used to either enter into or increase the A register or the Q register by the value in Δ . This value is signed allowing numbers of the magnitude plus or minus 127.

Example:

LDA	=N22	Loads A with 16_{16}
ENA	22	So does this
SUB	=N$1	Decreases A by 1
INA	-1	So does this

Where applicable, these instructions should be used in place of storage reference class with constant mode as these take only one word and 1.1 microseconds to execute.

The value in the address field is placed into Δ by the assembler. The ENA 22 machine instruction equivalent is 0A16.

5.5.2 Instructions SPB and CPB

These instructions are used to either set or clear the protect bit (Bit 17) in memory. For these instructions Δ is not used. The address of the core location which will have its protect bit either set or cleared must be in the Q register. If the program protect switch on the programmer panel is on, then these instructions must be protected. Otherwise, a program protect fault (interrupt on line 0, protect fault indicator on panel) will occur and these instructions become no operations.

Example: Clear the protect bits in core from 1000 to 2000.

```
TEMP        0           0
START       LDQ         =N$1000
LOOP        CPB                        Address is in Q
            INQ         1
            STQ*        TEMP
            ADQ         =N-$2000
            SQZ         DONE-*-1       Finished?
            LDQ*        TEMP           No
            JMP*        LOOP
DONE        -           -              Yes
            -           -
```

This particular routine is further simplified with the use of interregister class instructions.

Problem: The CLRPB subroutine clears protect bits on what core area?

Calling Program

```
                RTJ         CLRPB
        LWA     NUM         $4000
        FWA     NUM         $2000
```

Subroutine

```
        CLRPB            0             0
                         STQ*          TEMPQ+1
                         STA*          TEMPA+1
                         LDA           =X$7FFF
                         SUB*          (CLRPB)
                         RAO*          CLRPB
                         LDQ*          (CLRPB)
                         SOV           0
        LOOP             CPB
                         INQ           1
                         AAQ           0
                         SOV           DONE-*-1
                         JMP*          LOOP
        DONE             RAO*          CLRPB
        TEMPQ            LDQ           =NO
        TEMPA            LDA           =NO
                         JMP*          (CLRPB)
```

5.5.3 Instructions EIN and IIN

These instructions are used to either enable the interrupt system or inhibit the interrupt system. If the program protect switch on the programmer's panel is on, these instructions must be protected. Otherwise, a protect violation will occur. The interrupt system is inhibited immediately upon execution of the IIN instruction. However, for the EIN instruction one free instruction is allowed before the interrupt system becomes enabled.

Example:

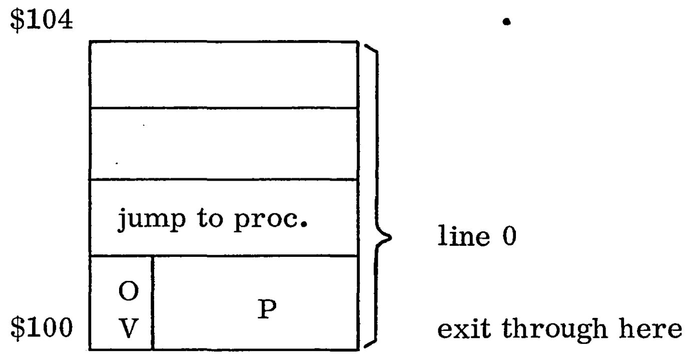
```
MAIN            RTJ             SUB
                -               -
                -               -
SUB             0               0
                IIN                             Interrupt System inhibited
                -               -
                -               -
                EIN
                JMP*            (SUB)           This instruction is free.
```

This subroutine will operate on any level without interference from any other level since the whole subroutine functions with the interrupt system off. Control is returned to the main program before the interrupt system is activated through use of the one free instruction following the EIN.

For example, if an internal interrupt occurred on line 0, P and the overflow indicator of the interrupted program would have been saved in word 0 of the trap for line 0 (location $100). Control would have been transferred to word 1 (location $101). If word 1 containeda jump tothe Internal Interrupt Processor, that routine would be executed. It would determine the cause of interrupt (program protect violation or parity error) by using the appropriate instructions (skips). Then the routine could exit back to the interrupted program. It would do that by:

```
EXI          00
```

The delta in the EXI instruction should be the lower 8 bits of the word 0 trap location for the appropriate line. In this case, the 00 means the trap for line 0, at address $100.



The EXI instruction (Exit Interrupt State) is used to exit from an interrupt subroutine. It restores the overflow indicator to its previous state, resets P of the interrupted program, and enables the interrupt system.

5.5.4 Instructions INP and OUT

The INP and OUT instructions are used for all input and output operations on the 1700. They are used to input or output data to or from the A register. They output function codes to the peripheral equipment from the A register, and they input status conditions of the equipment to the A register. The Q register is used to address the desired equipment. A brief introduction to I/O using these instructions is contained in Chapter 7.

5.5.5 Instructions SLS and NOP

The SLS or selective stop instruction is dependent upon the positioning of the selective stop switch on the programmer's panel. If the selective stop switch is up, then the program will stop on this instruction. If the selective stop switch is not up, then this instruction is the same as a NOP or No Operation Instruction, where the computer simply steps past this instruction without performing any operation. If the computer stops, program execution will continue by momentarily setting the RUN-STEP switch on the programmer's panel to the RUN position.

Example:

```
START          LDA          =N$1000
               NOP                              Put in for future expansion
               NOP
               -
               -
               -
               SLS                              Program stops if stop
               -                                   switch is up
               -                                Continues when run switch
               -                                   is hit or if stop switch
               -                                   is not up.
```

Problem:

The following is a conversion routine which converts a positive or negative hexadecimal number in the A register to the ASCII codes for the decimal number. It consists of a CONVRT subroutine and a main program CONTST which was used to check it out.

Study the program carefully, see how the conversion is done and how the parameters are passed.

This should be considered a final examination over the 1700 instructions and their use.

5.5.5

```
0001                            NAM     CONVRT
0002 P0000 0001                 BSS     SAVEQ(1),SAVEI(1),SAVEA(1)
     P0001 0001
     P0002 0001
0003 P0003 0003  BUF            BSS     BUF(3)
0004 P0006 0006  BUF1           BSS     BUF1(6)
0005 P000C 002B  SIGN           NUM     $2B,$2D
     P000D 002D
0006 P000E 0030  TAB            NUM     $30,$31,$32,$33,$34,$35,$36,$37,$38,$39
     P000F 0031
     P0010 0032
     P0011 0033
     P0012 0034
     P0013 0035
     P0014 0036
     P0015 0037
     P0016 0038
     P0017 0039
0007                            ENT     CONVRT
0008 P0018 0000  CONVRT         0       0
0009 P0019 48E6                 STQ*    SAVEQ
0010 P001A E0FF                 LDQ-    I
0011 P001B 48E5                 STQ*    SAVEI
0012 P001C 0842                 CLR     Q
0013 P001D 40FF                 STQ-    I
0014 P001E 0122                 SAP     POS
0015 P001F 0D01                 INQ     1
0016 P0020 0864                 TCA     A
0017 P0021 EAEA  POS            LDQ*    SIGN,Q
0018 P0022 48E8                 STQ*    BUF1+5
0019 P0023 E8EA                 LDQ*    TAB+0
0020 P0024 48E1                 STQ*    BUF1
0021 P0025 E000                 LDQ     =N$20
     P0026 0020
0022 P0027 48DF                 STQ*    BUF1+1
0023 P0028 48DF                 STQ*    BUF1+2
0024 P0029 48DF                 STQ*    BUF1+3
0025 P002A 48DF                 STQ*    BUF1+4
0026 P002B 0842  LOOP           CLR     Q
0027 P002C 0106                 SAZ     OUT
0028 P002D 3000                 DVI     =N10
     P002E 000A
0029 P002F EADE                 LDQ*    TAB,Q
0030 P0030 49D5                 STQ*    BUF1,I
0031 P0031 D0FF                 RAO-    I
0032 P0032 18F8                 JMP*    LOOP
0033 P0033 40FF  OUT            STQ-    I
0034 P0034 0C05                 ENQ     5
0035 P0035 CAD0  BACK           LDA*    BUF1,Q
0036 P0036 0FC8                 ALS     8
0037 P0037 0DFE                 INQ     -1
0038 P0038 8ACD                 ADD*    BUF1,Q
0039 P0039 69C9                 STA*    BUF,I
0040 P003A D0FF                 RAO-    I
0041 P003B 0DFE                 INQ     -1
```

```
0042 P003C 0171                SQM        DONE
0043 P003D 18F7                JMP*       BACK
0044 P003E E8C1    DONE        LDQ*       SAVEQ
0045 P003F C8C1                LDA*       SAVEI
0046 P0040 60FF                STA-       I
0047 P0041 C000                LDA        =XBUF
     P0042 0003 P
0048 P0043 1CD4                JMP*       (CONVRT)
0049                           END
```

```
I       00FF  SAVEQ   0000P SAVEI   0001P SAVEA   0002P BUF     0003P
BUF1    0006P SIGN    000CP TAB     000EP CONVRT  0018P POS     0021P
LOOP    002BP OUT     0033P BACK    0035P DONE    003EP
```

```
CONTST  2BB5
CONVRT  2BC8
```

TTY Output

```
PP
*
MI
*P
J
*ASSEM
OPTIONS LX
J
*P .
J
*L,5
J
*X,,
+14272 ←—— Answer
J
```

5.5.5

```
0001                              NAM      CONTST
0002                              ENT      CONTST
0003                              EXT      CONVRT
0004 P0000 0000    CONTST         0        0
0005 P0001 C000                   LDA      =N$37C0
     P0002 37C0
0006 P0003 5400  X                RTJ      CONVRT
     P0004 7FFF  X
0007 P0005 6800                   STA      WRITE+6
     P0006 0007
0008 P0007 54F4    WRITE          RTJ-     ($F4)
0009 P0008 0C01                   NUM      $0C01
0010 P0009 000F  P                ADC      COMPL
0011 P000A 0000                   NUM      0
0012 P000B 18FC                   NUM      $18FC    Writes answer
0013 P000C 0003                   NUM      3
0014 P000D 0000                   ADC      0
0015 P000E 14EA                   JMP-     ($EA)
0016 P000F 0161    COMPL          SQP      EXIT
0017 P0010 18FF                   NUM      $18FF
0018               EXIT           EXIT
0018 P0011 54F4
0018 P0012 0A00
0019                              END      CONTST
```

```
I         00FF   CONTST   0000P  WRITE    0007P  COMPL    000FP  EXIT     0011P
CONVRT    0004X
```

5.6 EXERCISES

5.6.1 Exercises - Constant Mode of Addressing

1. What is in the A and Q registers when location NEXT is reached?

```
a)                 --
                   LDA          =N$1F0C
                   ALS          3
                   LDQ          =N277
                   LRS          12
                   EOR          =N$0106
         NEXT      --- .        ---

b)                 LDA          =N$7FFF
                   LDA          =N6
                   MUI          =N6
         NEXT      ---          ---

c)                 LDQ          =AAB          See Appendix E for
                   LDA          =AXY          ASCII conversions.
                   AND          =ACD
                   LLS          16
                   AND          =AWZ
         NEXT      ---          ---
```

5.6.2 Exercises - Absolute Mode of Addressing

1. Write code to increase the contents of Index Register I by 10_{16}.

2. If C017 is in core location 3F, and 4016 is in core location 4017, where would program control be given for:

```
a)                 JMP-         ($3F)
b)                 JMP-         $3F
c)                 JMP+         ($3F)
```

3. What is wrong with these:

```
a)                 LDA-         0
b)                 LDA+         $7F3-$700
c)                 LDA+         (TEST)
```

5.6.3 Exercises - Relative Mode of Addressing

1. Will these instructions address backward or forward, and how many locations from P?

   a) 1831
   b) 1800
      EFFF
   c) 18FE

2. What modes of address are these assembly language instructions?

```
a)          LDA          TASK
b)          JMP-         ($44)
c)          RAO+         (HIND)
d)          STA*         *+3
e)          AND          =X$4111-4111
f)          STQ          (DRUM)
g)          LDA          =XTWIX
h)          LDA-         10
```

3. Why is relative addressing never used to address into the communications region?

5.6.4 Exercises - Indexing

1. What is wrong with these:

```
a)          LDA          AB,Q,I
b)          MUI          A,Q
c)          ALS          6,Q
```

2. What number is in A when NEXT is reached:

Core Location	Contents
---	---
0022-	0000
0023-	8024
0024-	0023
---	---
---	---

```
a)               LDQ         =N$-1
                 LDA-        ($23),Q
    NEXT         ---         ---

b)               LDQ         =N$24
                 LDA-        ($22),Q
    NEXT         ---         ---

c)               LDQ         =N$1
                 LDA         $23,Q
    NEXT         ---         ---
```

3. What is wrong with this program:

```
                 NAM
    START        LDA         =N$10
                 STA-        I
                 LDA+        $2000
    LOOP         SUB+        $2000,I
                 LDQ-        I
                 ADQ         =N-1
                 SQZ         DONE-*-1
                 JMP*        LOOP
    DONE         ---         ---
                 ---         ---
```

5.6.5 Shift and Skip Instructions

1. What will these machine language instructions do:

a) 0FF0
b) O1A3
c) OF52
d) OF88

2. What will be in the Q and A Registers when this program jumps to OVER.

```
Location     Opcode      Address       Comments

START        LDA         =N$738F       Loads 738F in A
             LDQ         =N$01CA       Loads 01CA in Q
             LLS         3
             QRS         14
             ALS         5
             SQM         GO-*-1
             ARS         2
             SAM         GO-*-1
             ALS         2
             QRS         3
GO           JMP         OVER
```

5.6.6 Review Exercises

1. Switch the contents of Location 1000 into Location 1001, and vice versa:

```
TEMP         0           0
             -           -
             LDA+        $1000
             STA*        TEMP
             LDA+        $1001
             STA+        $1000
             LDA*        TEMP
             STA+        $1001
             -           -
             -           -
```

Several points of note:

1. The problem descriptions seldom, if ever, specify core locations in decimal; they're assumed to be hexadecimal since rarely is a core location referenced decimally. Therefore, 1000 and 1001 in the problem mean 1000 hex and 1001 hex.

2. The TEMP location is necessary for switching to hold the one operand while the other is being switched. This form TEMP 0 0 can be used to simply define a core location; one cell is reserved and zero's placed therein.

3. The one-word relative mode of addressing is used for locations within the range of ± 127 to save one core location and one cycle time, and also to allow the program to be run anywhere.

4. Two-word absolute mode of addressing is used for the core locations 1000 & 1001. This is because this program could be finally loaded any number of core locations away from 1000 and 1001; therefore, a two-word instruction is needed. But why use absolute mode instead of relative? For programming "run anywhere" programs there are two basic rules:

   a) Everything that will move with the program is to be coded using relative mode;
   b) Everything that remains fixed in core is to be coded using absolute mode.

   Since the statement of the problem states the two fixed core locations to be switched regardless of where the program doing the switching is to be loaded, references to 1000 and 1001 should be made using absolute mode.

2. Transfer the contents of core locations 1000 through 1FFF to 3000 through 3FFF.

```
START           LDQ          =X$1FFF-$1000
LOOP            LDA+         $1000,Q
                STA+         $3000,Q
                INQ          -1
                SQM          DONE-*-1
                JMP*         LOOP
DONE            -            -
                -            -
```

3. Do a reverse transfer of problem 2, i.e., place contents of 1000 in 3FFF, 1001 in 3FFE, etc.

```
START           LDQ          =X$1FFF-$1000
                CLR          A
                STA-         I
LOOP            LDA+         $1000,I
                STA+         $3000,Q
                RAO-         I
                INQ          -1
                SQM          DONE-*-1
                JMP*         LOOP
DONE            -            -
                -            -
```

For this problem, two index registers are needed, one indexing up (I), and one indexing down (Q).

4. Do Example 2 without Index Registers:

```
                NAM          EXAMP
AREA1           ADC          0
AREA2           ADC          0
COUNT           NUM          0                      NUM sets the number 0
                LDA          =X$7FFF-$1FFF+$1000
                STA*         COUNT
                LDA          =N$1000                First Address in AREA1
                STA*         AREA1
                LDA          =N$3000                First Address in AREA2
                STA*         AREA2
LOOP            LDA*         (AREA1)                Indirect Addressing
                STA*         (AREA2)
                RAO*         AREA1
                RAO*         AREA2
                RAO*         COUNT
                SOV          DONE-*-1
                JMP*         LOOP
DONE            -            -
                -            -
```

Several points to note:

1. The number prestored in location COUNT is the difference between the largest possible positive number 7FFF and the difference between the beginning and end of the core blocks. If we increase this count by one each time through the loop, we will encounter an overflow condition when the count changes from 7FFF to 8000. We can then loop through and exit from the loop using this preset count.

2. AREA1 and AREA2 are preset by using the ADC pseudo-op to 0. Formerly the form AREA1 0 0 was used to perform the same function. The assembler, when encountering a zero in the opcode field, treats it as an ADC pseudo-op so both forms are equivalent.

3. This method of solving the problem is absolutely dependent on the use of indirect addressing since the addresses must be contained and manipulated in core cells.

CHAPTER VI

PSEUDO OPS

CHAPTER VI - Pseudo Ops

6.0 INTRODUCTION

The 1700 has 3 assemblers:

- Basic assembler, which operates as a stand-alone system in a 4-K computer
- Utility assembler, which operates under the utility system and requires an 8-K computer
- Macro assembler, which operates under the mass storage operating system and requires a 12-K computer

The standard pseudo ops covered in this chapter are available under all 3 assemblers, with the exception of some additional pseudo ops available only under the macro assembler. These are noted where they are described.

In addition to the mnemonics for the machine instructions which we have covered, there are certain instructions that are only recognized by the assembler. They are used by the assembler itself to control the assembly, control the data, reserve storage, convert data, signify beginning and end of assembly and control the output listing. These instructions are called pseudo instructions or pseudo ops.

6.1 NAM

The first instruction on any source program must be the pseudo instruction NAM. Its use is to signal the assembler when to begin assembly and how to set up its program counter for this assembly. Its form is:

Location	Opcode	Address	Comments
	NAM	Name	

If the location field is blank, the assembler will begin assembly with its counter at zero and signal to the loader in the object program that this program is program relocatable. This provides the ability to have the source program assemble without regard to where the program will finally be loaded and run in core. If a hexadecimal number appears in the location field, the assembler sets its program counter to that value and assembles the program absolutely. When a program of this type becomes loaded, it is loaded beginning at this absolute address specification. In the address field of the NAM pseudo op will be the program name which is reproduced on the output listing.

Example:

```
                NAM         SORT
                -
                -
                -
```

This program is assembled program relocatable and can be loaded anywhere into core. The name SORT will appear on the output listing.

```
       $100     NAM         INTERRUPT
                -
                -
```

This program is assembled absolutely, beginning at 100 hex, and will be loaded into core at 100 hex only.

6.2 END

The last instruction in the program must be an END pseudo instruction. It marks the physical end of the program. The address field may contain an entry point to the program. This is called a named transfer address, and it is the entry point where it is desired for execution to begin after the object program is loaded.

Example:

```
                NAM         SORT
                ENT         START
       START    -           -
                -           -
                -           -
                END         START
```

6.3 ENT, EXT

Two pseudo instructions are used to provide communication between programs. They are the ENT (entry point) pseudo instruction, and the EXT (external point) pseudo instruction. Those locations internal to a program that are needed in another program are declared as entry points to the immediate program. The other program can then refer to these entry points in its immediate program by declaring them as external. The names must match identically. Since communication between these two programs can not be made at assembly time (since both programs can be assembled at different locations at different times), the relocating linking loader must provide the correct location addresses when these two programs are finally loaded together. To accomplish this, the loader builds loader symbol tables (Figure 13) where it places references to all entry points and external points. These tables locate exactly for the loader where the entry point addresses are and where the external point references are that need patching with their corresponding entry point addresses. When it finds a match in names between an entry point and an external point, it does the patching.

Example:

Program 1 is written and it needs to extract data from program 8 which has not yet been written. The writers of both programs agree to a 10 word area with the name of CLARK. Since program 1 needs to refer to CLARK in program 8, it declares CLARK as an external. This external declaration allows the assembler the use of this symbol which is not otherwise defined in program 1.

```
NAM       PROG1
EXT       CLARK
-
-
-
LDA+      CLARK
```

Program 8, when it is finally written, will declare CLARK as an entry point.

```
NAM       PROG8
ENT       CLARK
-         -
-         -
-         -
BSS       CLARK(10)
-         -
```

When both of these programs are finally loaded together in core, the loader will link the address of CLARK at program 8 to its correct reference in program 1.

The EXT described above is called an absolute external. It means references made from the program to the external are assembled in absolute form (even if a relative mode is used in the instruction).

More than one symbol can be defined with each ENT and EXT instruction, since their general form is:

EXT $n_1, n_2,$ ----

ENT $n_1, n_2,$ ---

n - name

6.4 EXT*

Another form of external is available under the macro assembler: the relative external. It causes the loader (at load time, when linking is done) to patch in the relative distance from the referencing instruction to the location of the external, rather than the absolute core location. This allows the use of relative references.

```
Example:             EXT*        TAG
                      |
                      |
          5800       RTJ         TAG
          7FFFX       |
                      |
```

A two-word relative mode must be used in referencing these externals.

6.5 EQU

It is common to use symbols in place of constants or known address locations. The EQU pseudo instruction provides a means of declaring to the assembler the equivalence of a symbol with a number or expression. The form of EQU is:

Location	Opcode	Address	Comments
	EQU	ONE(1), TWO(2), THREE(3)	

The symbol with its equivalent number is placed in the assembler's symbol directory and all references to that symbol will yield its equivalent number. It is important to note that the EQU does not generate any code; it simply tells the assembler another value for symbols found in the program. For example, it uses a 2 wherever it sees the name TWO.

Example: Count the number of times the exact bit configuration 1110 appears in bit positions 4 to 7 of core locations 1000 to 10CE.

```
          NAM     FIND
COUNT     0       0
          EQU     MASK1($00F0), MASK2($00E0), FIRST($1000)
          EQU     LAST($10CE)
START     LDQ     =XLAST-FIRST
LOOP      LDA+    FIRST,Q
          AND     =NMASK1        And out all but bits 4 thru 7.
          EOR     =NMASK2        Look for exact match.
          SAN     OVER-*-1       Was match not exact?
          RAO*    COUNT          No, match was exact.
OVER      INQ     -1             Yes, no match.
          SQM     DONE-*-1       Finished?
          JMP*    LOOP           No.
DONE      -       -              Yes.
          -       -
```

By using the EQU, the same general problem with different parameters could be solved with this program simply by changing the EQU card. Assume the problem looked for 10110_2 in bit positions 8 through 12 of core location 3020 through 3F21. Simply pull out the EQU card and insert one:

```
EQU         MASK2($1F00), MASK2($1600), FIRST($3020)
EQU         LAST($3F21)
```

The EQU instruction is especially useful for referencing the mask tables in low core. (See Appendix I.) These masks are available for foreground or background programs to use, rather than defining additional core locations in a program to contain masks. The EQU's to be used would be as follows:

```
EQU         LPMASK($2), NZERO($12), ZERO($22)
EQU         ONEBIT($23), ZROBIT($33)
```

These are the same EQU's used by the system and they make it easier to remember which mask is being used. For example:

```
LDA-        LPMASK+2
```

This can be used instead of:

```
LDA-        $4
```

The same code is generated: C004

It is easier to remember that LPMASK+2 is a mask location containing two one bits on the right end than to remember what location $4 contains. NZERO+4 would contain 4 zero bits on the right. Location ZERO always contains a 0 word. ONEBIT+5 would contain a one bit in bit position 5; ZROBIT+8 would contain a zero bit in bit position 8.

6.6 NUM

In order to insert known constants into the assembly, the pseudo op NUM is used. Its form is:

Location	Opcode	Address	Comments
s	NUM	$c_1, c_2, \ldots, c_n$	

s means a symbol

16-bit constants are inserted, in line, one constant to a word. If a symbol is specified in the location field, it is assigned the storage address of the first constant.

Example:

```
                NAM         EXAM
      HERE      NUM         $7312,21,-21,-$216
```

Since this is a program relocatable assembly (blank in location field of NAM card), HERE is at program counter P0000 and the following constants are inserted:

```
      P0000-    7312
      P0001-    0015
      P0002-    FFEA
      P0003-    FDE9
```

Expressions are not allowed.

```
                NUM         $7312-41          Illegal
```

6.7 ADC

To insert in line a table of addresses, the ADC pseudo instruction is used. It functions identically to the NUM pseudo instruction except expressions may be used and the result is evaluated for only 15 bits since an address value cannot exceed 15 bits in length. Bit 15 will be set if the expression is enclosed in parentheses (indicating an indirect reference). Its form is:

s ADC e_1, e_2---e_n

Example:

```
   Location     Opcode      Address           Comments
   --------     ------      -------           --------
                NAM         EXAM
                EQU         TEN(10)
                EQU         MASK($F302)
                =
   HAT          ADC         TEN               TEN is 000A   (See EQU)
                =
   STILL        ADC         (HAT)             Will set Bit 15
```

Assume the program counter for HAT is at P0102 and for STILL, P01FF; then:

```
   P0102        000A

   P01FF        8102
```

6.8 ADC*

Under the macro assembler there is a second form of the ADC pseudo op, the ADC* pseudo op. This form functions identically to the ADC pseudo op in the utility assembler. However, all address expressions evaluated are then placed in relative form. Examples of both forms are on the following page.

```
                NAM             EXAM1
                BSS             TAG(10) ,TAG1(10)
HERE            ADC             TAG ,TAG1
```

HERE which is at P0014 has the absolute address of TAG (P0000), and HERE+1 (P0015) has the absolute address of TAG1 (P000A).

```
                NAM             EXAM2
                BSS             TAG(10) ,TAG1(10)
HERE            ADC*            TAG ,TAG1
```

HERE, at P0014, has the relative address of TAG (FFEB, or twenty decimal locations back) and HERE+1 (P0015) has the relative address of TAG1 (FFF5).

6.9 ALF

ASCII characters are stored in consecutive locations, two 8-bit characters for each core location, by the ALF pseudo instruction. The ALF pseudo instruction is used to pack a core area with a message which can be used for subsequent output to ASCII devices like the teletype. A symbol, if used in the location field, will refer to the first word of the block. The format for the ALF pseudo instruction is:

```
s               ALF             n, <2n characters>
```

Example:

```
                NAM             EXAM
HERE            ALF             3,ABACAD
```

Three words are packed with the ASCII equivalents of ABACAD:

```
P0000           4142
P0001           4143
P0002           4144
```

Table of ASCII equivalents is found in Appendix E.

A blank is stored into unused locations.

Example:

```
          HERE            ALF             6,ABACAD

produces: P0000           4142
          P0001           4143
          P0002           4144
          P0003           2020
          P0004           2020
          P0005           2020
```

6.9

The ALF pseudo op in the utility system only allowed specification for its message by the use of an unsigned integer for the number of core locations to be reserved. In the macro assembler a second form for this pseudo op is available:

```
                    ALF          n, <message>n
```

n may be a non-integer character which signals the end of the message. n is a delimiter and appears before the comma and after the message.

This form is an advantage where the programmer does not desire to count the number of words in his message and will find no conflict between his message and the terminating character used.

For either form the pseudo op ALF will pack two ASCII characters per word. The address of the first location of the message in core will be assigned to the symbol in the location field, if specified.

Example:

```
                    NAM          EXAMP
      GO            ALF          Z,DATAZ
```

Two words are reserved, starting at location GO for the ASCII equivalent of DATA.

6.10 DEC

A DEC pseudo op is available under the macro assembler. Suppose the following problem needed to be solved:

$$y = \frac{1.63 \times 10^3}{.0074 \times 10^6} x^2 + 21246 \times 10^{-2} x + \frac{81 \times 2^6}{.11 \times 10^3}$$

Insertion of the constants in this problem would be facilitated by having the assembler do the binary or decimal conversions. The DEC pseudo op allows insertion of constants with decimal or binary scaling factors. Its form is:

s   DEC $k_1, k_2 \ldots, k_n$

k is a constant - fDdBb

Example:

```
                    NAM          EXAMP
      HERE          DEC          163D1,74D2,21246D-2,81B6,11D1
                    -
                    -
```

This example shows the constants from the equation inserted. The decimal numbers are converted mentally to integers. HERE is the symbolic address of the first of these decimal constants which are inserted one per core cell. The size, then, of the converted decimal constants must lie within the range of ±32,767.

6.11 VFD

The VFD pseudo op is available under the macro assembler. It is frequently desirable to pack data into memory locations. The VFD (variable field definition) pseudo instruction assigns data to consecutive locations in the instruction sequence without regard for computer words. Data is stored in bit strings rather than word units. Its format is:

$$s \qquad \text{VFD} \qquad m_1n_1/v_1, m_2n_2/v_2, \ldots, m_nn_n/v_n$$

m will specify the mode. Three modes are possible:

N	numeric constant
A	ASCII character code
X	expression

n will specify the number of bits and v is the value. n may be 16 bits or less for either N mode or X mode; however, for A mode n must be some multiple of 8 since ASCII character conversion is meaningless for a non-multiple of 8 bits. Numeric constants must be within the range of ±32,767.

Example using numeric constants:

```
            NAM         EXAMP
TAG         VFD         N4/$F,N8/6,N8/-6,N2/16
            LDA
            -
```

The assembly of the bit strings begins with the high order bit of the first core cell (in this example the core cell labeled TAG). TAG gets packed with 1111 which is the binary equivalent of hexadecimal F. The next 8 bits of TAG get packed with the binary equivalent of 6 which is 0000 0110. The next 8 bits (which will now be the lower four bits of TAG) and the upper four bits of TAG+1 get packed with -6 or 1111 1001. The next two higher order bits get packed with as much of the number 16 as is possible, that is, with the low order two bits (00). The remaining bits of TAG+1 set to zeros. TAG and TAG+1 will then look like this:

```
TAG         F06F-       1111  0000  0110  1111
TAG+1       9000-       1001  0000  0000  0000
```

If the number of bits specified is not sufficient for the value then the high order bits of the value are truncated (chopped off), as many as are necessary. If the number of bits is larger than the value, then the sign of the value is extended.

Example using expressions:

```
            NAM         EXAMP
            EQU         TAG($4FF1),HAT(20)
TOP         VFD         X8/TAG+2,X8/HAT
```

The previous example shows the use of expressions where the expression is evaluated absolutely (since neither TAG nor HAT is relocatable). If fewer than 16 bits are specified, the absolute expression by itself is evaluated (using 16 bits) and is truncated. The previous example is decoded by the assembler.

```
          TOP          F314
```

When the expression is evaluated relatively, the n must be 15 and the expression must be positioned so that it will be stored right justified at bit position 0 of the computer word.

Example using ASCII character mode:

```
                       NAM          EXAMP
          TAP          VFD          A24/ABC,N8/$3F
```

The above example illustrates three ASCII characters, A, B and C; these will be converted using 8 bits for each, followed by the numeric constant 3F hex in the lower 8 bits of TAP+1. The above example is decoded by the assembler.

```
          TAP          4142
          TAP+1        433F
```

6.12 BSS, BZS

Blocks of data storage can be allocated within the program using either the BSS or the BZS pseudo instructions. The block is given a name and a size according to the following format:

BSS $n_1(s_1), n_2(s_2), ---n_m(s_m)$

n is name of block
s is size

These pseudo instructions reserve areas. The BZS area is zeroed out at load time while the BSS block is not changed at load time; therefore, anything may be initially set in a BSS block at run time.

Example:

```
                       NAM          EXAM
                       BSS          AA(10),BB(20)
                       LDA          -
                       -            -
```

The symbol AA will be assigned the address of the first location of the block of 10 and the symbol BB will be assigned the address of the first location of the block of 20 locations reserved. When this program is loaded, anything can be initially contained in these first 30 locations. Had a BZS been used instead of the BSS, these first 30 locations would have been zeroed at load time.

6.13 DAT, COM

Two other pseudo ops are used to reserve areas for use that are outside the bounds of the main program. These areas are reserved by the DAT and COM pseudo ops. Refer to Figure 13. Notice that the common storage area (reserved by the COM pseudo instruction) is the area that is used by the loader. This area cannot be preset with data and is used only at run time when the loader becomes destroyed. The data area (reserved by the DAT pseudo instruction) is assigned an area with the programs themselves; in fact, the data block will precede the program that declares it. The data area can be preset. The loader will make common and data area assignments just once and will use its common counter and data counter at this assigned value for the rest of the programs loaded. It is necessary, then, for the first subprograms of a run declaring common or data storage to declare the largest amount necessary. The format for the COM and DAT pseudo instructions is the same as for the BSS and BZS and is:

DAT $n_1(s_1), n_2(s_2), \ldots n_m(s_m)$

COM $n_1(s_1), n_2(s_2), \ldots n_m(s_m)$

Example:

```
NAM     EXAM
BSS     AA(30)
DAT     CAT(40) , RAN(20)
COM     CCC(40) , AB(10)
-       -
-       -
```

The BSS will reserve 30 locations within the program while the DAT will reserve a total of 60 core locations, reserved outside of the program area. In fact, this data area will immediately precede the main program area in core. The common area, 50 words in this example, is reserved at the high end of core where the loader resides at load time. This common area cannot be preset with data and can only be used at run time, when the loader is no longer needed.

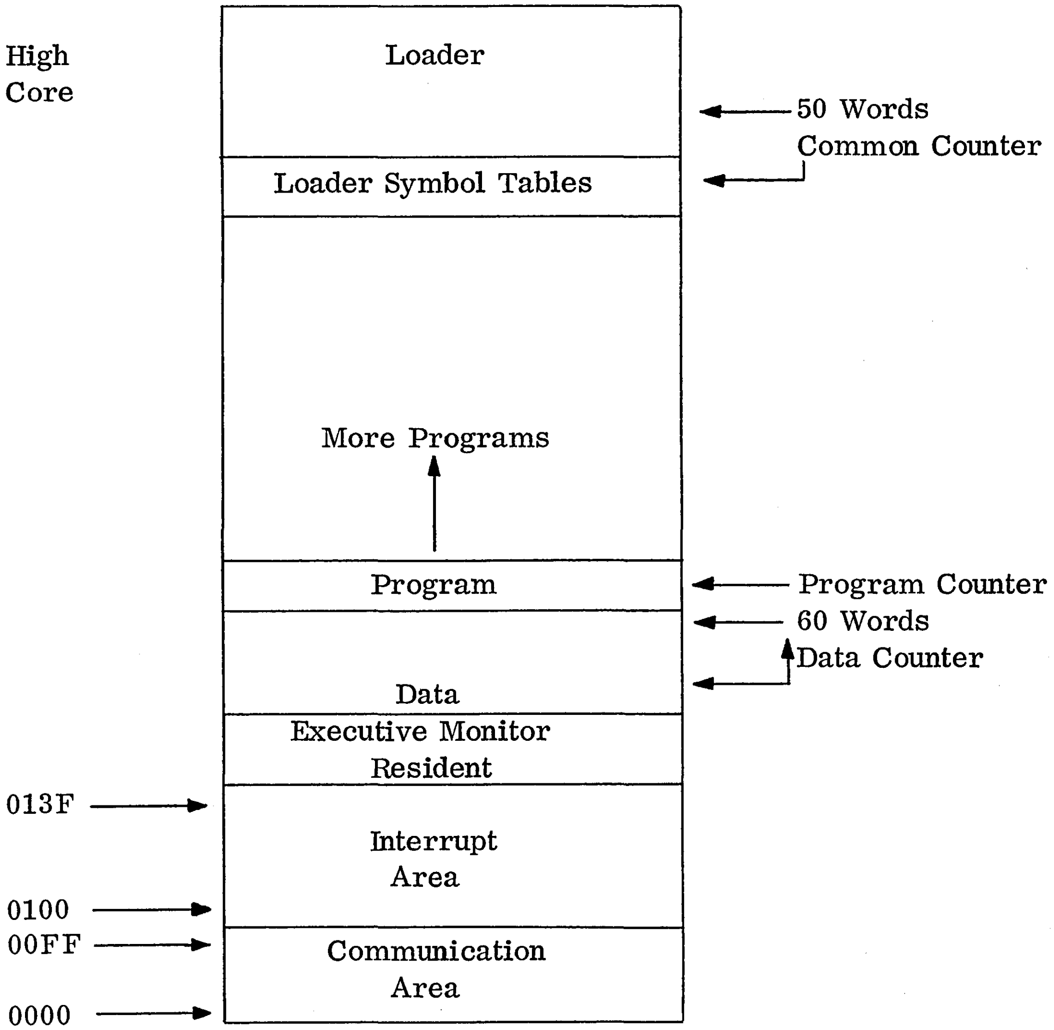


Figure 13. Data, Program and Common Counters.

Figure 13 illustrates the three counters: data, program and common counter. There are three types of relocation possible when loading programs, each type using its appropriate counter. References to addresses will be relocated using the data counter if the address is in the data area, the program counter if the address is in the main program area and the common counter if the address is in the common area.

Example:

```
                              NAM        EXAMP
                              DAT        AA(10)
                              COM        BB(10)
                              BSS        CC(10)
                              LDA+       AA+3
                              STA+       BB+7
                              STA+       CC+4
                              -
                              -
                              -
                              .
```

The listing for the above looks like:

```
001.                                NAM        EXAMP
002.                    0000D       DAT        AA(10)
003.                    0000C       COM        BB(10)
004.        P0000       000A        BSS        CC(10)
005.        P000A       C400        LDA+       AA+3
            P000B       0003D
006.        P000C       6400        STA+       BB+7
            P000D       0007C
007.        P000E       6400        STA+       CC+4
            P000F       0004P
```

The first column is the line number. The second column is the core location in hex. The P preceding it indicates that the value of the program counter will be added to the number at load time, yielding the actual core location.

Hex word followed by relocation symbol: P for program counter, D for data counter, and C for common counter.

Although there is only one common area and one data area assigned per core load, references to data in these areas can be made by all programs in core. The relative position with respect to the data or common counter for the data desired must be known by each program but the same names need not be used by different programs to reference the same data.

Example: Program 1 is the first program loaded. It must declare the largest data or common area.

```
                              NAM        PROG1
                              DAT        AX(100) ,BX(50) ,CX(100)
                              -
                              -
```

When the program is loaded, a data area of 250 locations is assigned and the data counter is set at the beginning of this area.

Suppose PROG6, which is the sixth program loaded, is interested only in the data which PROG1 knows as the 26th to 30th locations in BX.

```
NAM        PROG6
DAT        DUMMY(125),MINE(5)
--
LDA        MINE
---
---
```

DUMMY is not used by program 6. It only allows a skip past the AX and first 25 locations of BX corresponding to program 1. Reference to MINE in program 6 will yield the same data as reference to BX+25 in program 1.

6.14 ORG, ORG*

Presetting the data area is accomplished by the use of the ORG pseudo op. Its form is:

```
ORG        a
```

This pseudo op changes the value of the assembler's counter to agree with a. All instructions or data following the ORG instruction are assembled into consecutive locations until either another ORG instruction is encountered or an ORG* is encountered. When an ORG* is encountered, the assembler's program counter is set to the value that it would have been if the very first ORG instruction had not occurred. The address expression (a) may be either positive program relocatable, positive data relocatable or absolute. Notice common relocatable address expressions are not allowed since the common area cannot be preset.

Example:

```
NAM        EXAMP
DAT        AX(20) ,BX(40)
LDA        -
-          -
-          -               Assume P.C.=P0030
ORG        AX+10
-          -
-          -
ORG        AX
BZS        10
ORG*       -
LDA        -
-          -
```

In this example the program counter starts off at 0000 and proceeds in sequence until the first ORG instruction is assembled. Since the address expression refers to the data area, the assembler's program counter will now be at D000A, indicating that the code beneath this first ORG pseudo instruction will be inserted beginning in the 10th data area location. When the second ORG instruction is encountered, the code under it (BZS10) is assembled into the beginning of the data area. When the ORG* instruction is encountered, the program counter is set to P0031; this is the next location following the last location assembled before the first ORG instruction. Any symbols used in the address expression of the ORG pseudo op must have been previously defined in the program.

The following example can be used to illustrate presetting values in A and B in the DATA block:

```
A≡ 0000D                DAT       A(5),B(2)
X≡ 0000C                COM       X
C≡ 0007D                DAT       C(50)
   0000P     START      0         0
   0001P                LDA-      $FF
   0002P                STA+      X
                        ORG       A
   0000D                NUM       1,2,3,4,5,6,7
                                  \_______/\__/
                                      A      B
                        ORG*
   0004P                RAO-      $FF
```

6.15 IFA, EIF

The macro assembler contains a conditional assembly instruction. With the pseudo op IFA, it is possible to specify portions of a program to be either assembled or excluded during assembly time. Its format is:

```
s                IFA         e1,c,e2
```

(This pseudo op can be used within a macro skeleton.) e may be an expression and c specifies one of four conditions:

EQ	$e_1=e_2$
NE	$e_1\neq e_2$
GT	$e_1>e_2$
LT	$e_1<e_2$

The termination of the coding encompassed within the range of an IFA pseudo op is accomplished with the pseudo op EIF. Since nesting is allowed, a match between an IFA and EIF pseudo op range is made by correspondence between the first two characters of the symbols in the location field of the IFA and the address field of an EIF.

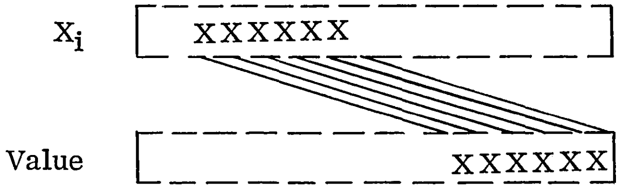
```
Example:              NAM      EXAMP
                      EQU      AB(10) ,AC(20) ,AD(30)
                      -
                      -
            TOTS      IFA      AB+AC,EQ,AD
                      LDA      TAG
                      STA      TAG1
                      EIF      TOTS
```

The LDA TAG and the STA TAG1 instructions will be assembled in this example since equality exists. Changing the EQU card, however, or changing the condition from EQ to NE would have excluded these two lines of code from the assembly.

Problem:

VALUE PROBLEM (COMMON)

Starting in the 11th location of COMMON are 10 words. Bits 13-8 of each of the10 words are to be compared with bits 5-0 of a location called VALUE which is external to this program. Do not destroy the original contents of VALUE. Count how many complete matches are found in the bit strings compared and store the answer in the 7th word of the data block. For example, if bits 5-0 of VALUE contain 101100 and bits 13-8 of X_1 contain 101100, one match has been found. Any other bit configuration would be a no-match. Write a complete program to solve this problem.



6.16 MAC, EMC

The macro assembler gets its name from the macro capability incorporated therein. An often used set of instructions may be grouped together to form a macro. Macros then need be defined once and thereafter the whole macro structure will be incorporated in line in the assembly generated coding whenever the macro name is called. Each macro has a name which is first defined by the use of the MAC pseudo op and thereafter the name can be placed in the opcode field as if it were an instruction or pseudo instruction and the assembler will substitute, starting at that location, the whole structure that was previously defined by that name. The macro must first be defined. The form is:

s MAC $p_1, p_2, \ldots, p_n$

s is the name of the macro. The p's are symbols of one or two characters that will define variables within the macro structure. Parameters are enclosed in apostrophes as shown within the macro. The keypunch code for apostrophe is 8-5. The macro structure itself is defined to be finished when the EMC pseudo op appears in the opcode field. EMC is always the last instruction in a macro definition.

Example:

```
                NAM        EXAMP
        HELP    MAC        XA,XB
                LDA        'XA'
                STA        'XA'+1
                ADD        'XA'-1
                STA        'XB'
                EMC
```

HELP is the name of the macro and XA and XB signify variables within the macro. All the code between the MAC and EMC is the macro structure. Anything in this structure can be made variable. In this assembly only positions of the address field were varied. Macros canbe of any length and they can also be nested. A macro must be defined before it is called by name. Calling the macro in the above example would look like this:

```
                -          -
                -          -
                -          -
                SAZ        OVER-*-1
                HELP       TAG,TAG1
        OVER    -          -
                -          -
```

The macro is called by placing its name in the opcode field. The assembler will search the symbol directory for a macro with the name of HELP, and if found, the complete macro structure is placed in line at this point. When calling a macro, the variables must be specified. For this example TAG will be inserted for each XA reference and TAG1 will be inserted for each XB reference. Effectively, then, the assembler will place in line the following:

```
                LDA        TAG
                STA        TAG+1
                ADD        TAG-1
                STA        TAG1
```

A macro could be defined as "an instruction which 'stands for' a number of other instructions."

6.17 LOC

Since the code for a macro is inserted in line wherever it is "called," if there are any symbols in the location field of the macro, it could be called only once, since the symbol would be doubly defined if the macro were called again. This problem can be eliminated by defining the symbols local to the macro.

Symbols that are local to the macro being defined are listed in the LOC pseudo op. This pseudo op immediately follows the MAC pseudo op and it allows use of one- or two-character symbols local to the macro so that the same symbols can also appear in the main program. Its form is:

LOC $s_1, s_2, \ldots., s_n$

Example:

```
                 NAM       EXAMP
        TOPP     MAC       AD,AB
                 LOC       G1,G2
                 LDA       'AD'
        'AB'     EOR       =N$0171
        'G1'     STA       'AD'+1
        'G2'     JMP       'AD'+2
                 EMC
```

Symbols passed as parameters may not be defined as local.

6.18 IFC

A conditional assembly pseudo op is available for use within the macro skeleton. It is the IFC pseudo op. It operates the same as the IFA covered previously; however, it has only two conditions, the NE and EQ. Its form is:

s IFC a_1, c, a_2

Each a must be a string of from one to six characters, or a formal parameter specified in the MAC statement. The character strings should not contain commas, blanks or apostrophies. Two character strings are equal when they contain the same characters in the same position and are of the same length. Characters in excess of 6 are ignored. Termination of the range of the IFC is made when an EIF is encountered with the first two characters of the symbol in its address field matching the first two characters in the location field of the IFC.

Example:

```
                NAM         EXAMP
      TOTAL     MAC         XA,XB,XC,XD
                LD'XA'      =N$4000
                STA         'XB'
      IT        IFC         'XC'EQ'XD'
                ADD         'XB'+2
                STA         'XB'
                EIF         IT
                EMC
```

If parameter XC and XD are equal when this macro is called then the ADD and STA instructions will be incorporated in line with the rest of the macro structure. If parameters XC and XD are not equal when the macro is called, these instructions will not be inserted in line as part of this macro structure during assembly.

```
                -
                -
                TOTAL       Q,TAG,TEN,NINE
                -
                -
                -
```

The generated code is:

```
                LDQ         =N$4000
                STA         TAG
```

Since TEN and NINE do not match character for character, the IFC condition is not met and the ADD followed by the STA instructions are not assembled.

Problem:

Write a macro for an INI instruction. Add a test routine to check it out.

The INI macro should increase the I Register by any constant exactly the same way the INA increases the A Register or the INQ increases the Q Register. In other words, no other registers should be destroyed when the macro is called.

6.19 NLS, LST, SPC

Three pseudo instructions are used for control of the listing. They are NLS, which prevents normal output list until a LST instruction is encountered or until the end of a program. Spacing paper on the printer is accomplished by the SPC pseudo op. The number of lines to space is specified as an absolute address expression in the address field.

6.19

Example:

```
          NAM       EXAMP
          LDA       -
          -         -
          SPC       12              12 lines are spaced
          -         -
          -         -
          NLS                       Listing is stopped
          -         -
          -         -
          LST                       Listing is enabled
          -
```

6.20 EJT

In addition to the NLS, LST, and SPC listing control pseudo instructions, the macro assembler also has an EJT pseudo op. This instruction causes page ejection during printing of the list output.

6.21 OPT

Three standard options determine the type of output from the assembler. All three are automatically selected if no OPT statement is encountered before the first NAM. OPT is the only pseudo instruction that may precede the NAM pseudo op. No code is generated by this pseudo instruction.

OPT must begin in card column 2.

Normal execution of assembly produces list output on the standard list device, punch output on standard punch device and load and go output on the mass storage device. These are options L, P and X. Two other options are available. The M option will enable listing of macro skeletons and an A option will cause abandonment of the assembly and will return control to the operating system. To exercise these options or to eliminate any of the three standard options, the OPT pseudo instruction is used. When the OPT pseudo instruction is encountered by the assembler it will type OPTIONS on the teletypewriter and allow the operator to manually reset the options desired. He can choose any or all of these five options.

Option	Meaning
L	List output on standard list unit
P	Punch binary output on standard punch unit
X	Load and go; executable output loaded on a mass storage device
M	List macro skeletons
A	Abandon assembly

Relocatable binary output is selected by the P option. The format is described in the 1700 Operating System Reference Manual.

If the X option is selected, relocatable binary output is placed on the mass storage unit for subsequent loading and execution as described in the 1700 Operating System Reference Manual.

The L option results in assembly listing.

6.22 MON

After the last subprogram has been assembled, control can be returned to the operating system by use of the MON pseudo op. It may be used only after the END statement.

MON must begin in card column 2.

Example:

```
cc
2
OPT
            NAM        PGM


            END        n ←———————— Name of main entry point in main
            NAM        SUB          program


            END          ←———————— No name on subroutine end card
MON
cc
2
```

6.23 EXERCISES ON PSEUDO OPS - UTILITY ASSEMBLER

1. What are the errors in this program?

```
Location     Opcode     Address          Comments

             NAM        EXAMP
LED          EQU        720
TAG          NUM        -72,$FFFF,72
TAG11        BSS        25
             EXT        LAD,TAG
START        LDA+       TAG
             STA+       LAD
             -
             -
             END        START
```

6.23

2. Why will the ORG LIST produce an error?

```
COM          LIST(30)
-
-
ORG          LIST
-
-
```

3. In this example:

```
COM          LIST(30)
-
-
LDA          LIST
```

this assembler will decode the LDA LIST as:

```
C400
0000C
```

Why?

4. What is the problem?

```
-
-
BSS          LIST(10)
-
-
LDA          LIST+3
SAZ          -
-
-
```

6.24 EXERCISES ON PSEUDO OPS - MACRO ASSEMBLER

1. What code is produced:

```
a)   VFD          N12/-17 ,N5/$7F2 ,N15/47
b)   VFD          A8/A,A8/B
c)   DEC          16D1B4
d)   ADC*         *-1
```

2. What instructions are assembled:

```
                -
                -
                EQU         AA(10), BB(20), CC(30)
        GO1     IFA         10, GT, BB-15
                LDA         =N$1000
        TO2     IFA         CC-AA, EQ, BB
                ADD         =N$1000
        HO3     IFA         40, NE, BB*2
                ADD         =N$2000
                EIF         HO
                EIF         TO
                EIF         GO
                -
                -
```

3. If this macro definition:

```
        PRINT   MAC         XA, XB, XC
        TAG     ALF         Z, ERROR'XB'Z
                IFC         'XA', EQ, LU
        TAG1    ALF         Z, LOGICAL UNIT'XC'Z
                EIF
                EMC
```

is called by

```
                PRINT       LU, 6, 8
```

what assembly language is produced when this macro is called?

4. What discrepancy is in this program?

```
                -
        FALL    MAC
                LOC         TE
        'TE'    NUM         $7FF3
                LDA         'TE'
                EMC
                -
                -
                FALL
                LDA         TE
                -
                -
```

CHAPTER VII

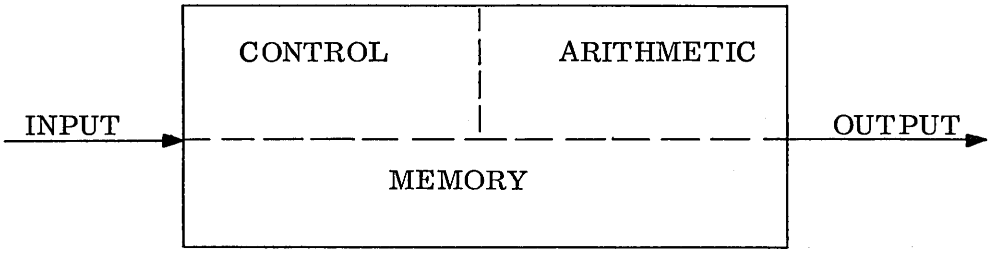
INTRODUCTION TO MACHINE LANGUAGE I/O

CHAPTER VII

Introduction to Machine Language I/O

7.0 INTRODUCTION

The 1700 is composed of a 1704 and various peripherals. The 1704 contains the registers and the logic necessary for bringing data into the computer, performing operations upon the data and sending the data out of the computer for future reference and/or display.



1704

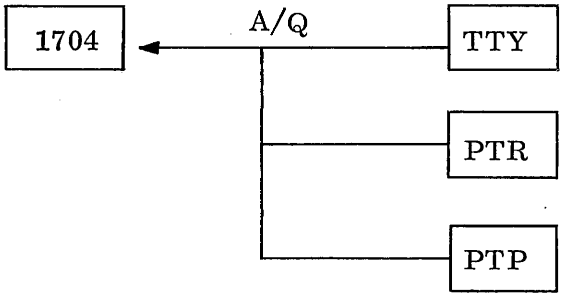
The peripherals are composed of devices capable of sending and/or receiving data. Such devices are the teletype, paper tape reader and paper tape punch. Many other peripherals are available and discussed in Chapters 12 and 14.

The peripherals and the 1704 cannot communicate directly; therefore, an interpreter is required. The interpreter is referred to as a controller. The program tells the controller the operation to be performed and the controller directs the peripheral in the performance of the operation.

7.1 UNBUFFERED I/O

7.1.1 Use of Registers in I/O

Communications among the 1704 and the peripherals is accomplished via one input/output (I/O) channel attached to each controller. The I/O channel works in conjunction with the 1704's A register and Q register. It is, consequently, called the A/Q channel.



7.1.1.1 Q Register

The Q register designates the equipment to be referenced and directs the operations to be performed. The Q register will be in the following format when performing an I/O operation via the A/Q channel.

	15 11	10 A 7	6 1	0
Q	W	E	S	D

The W field, bits 15-11, will always be zero except when referencing a 1706; this will be discussed later.

The E portion, bits 10-7, designates the equipment number being referenced. The equipment number will correspond directly with a hardware switch located on each controller. The number will vary from a hexadecimal 0 to a hexadecimal F.

The S portion distinguishes among peripherals attached to the same controller. The bits composing the station code will vary with controllers.

The D portion is the director bit or bits which designates the type of information being transferred: data, status or function. The number of bits used to compose the D portion also varies according to the controller being referenced.

7.1.1.2 A Register

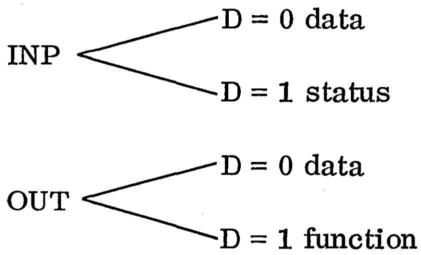
The A register sends and receives all communications between the 1704 and the peripherals; that is, the data, functions, or status.

7.1.2 Functions, Status, and Data

The 1704 is capable of sending or receiving data, sending a function, or receiving status. The D portion of Q and the I/O instruction executed denote which of the three operations is to be performed. All input/output operations via the A/Q channel are performed with two instructions.

INP
OUT

The INP instruction brings information into the A register: data or status. The OUT instruction sends information from the A register: data or function. Bit 0 of Q is usually the D portion, designating the type of information being transferred. (Note: the exceptions are discussed in Chapter 14.) When bit 0 of Q is a 0, the transfer of data is designated. The direction of the data flow is indicated by the I/O instruction. The INP brings data into A. The OUT sends data from A. When bit 0 of Q is a 1, the transfer of status or the transfer of a function is requested. INP requests status while an OUT sends a function.



7.1.3 Summary, Unbuffered I/O

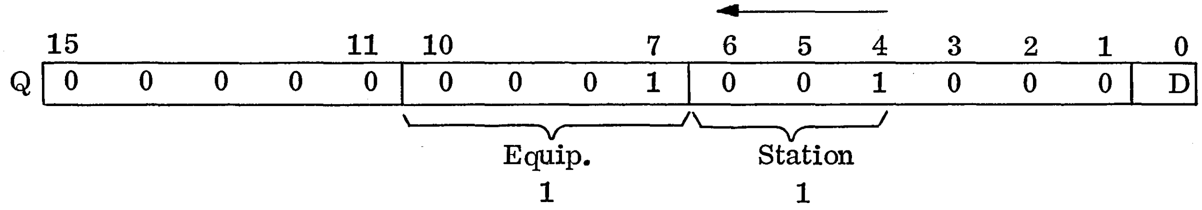
In review, all input/output operations performed by the 1704 will take place via the A/Q channel. The Q register indicates the peripheral being referenced and the type of information being transferred. The information to be transferred will be brought into or sent from the A register, depending upon the instruction executed: INP or OUT. Three types of information may be transferred: data, function, or status.

7.1.4 Low-speed Package

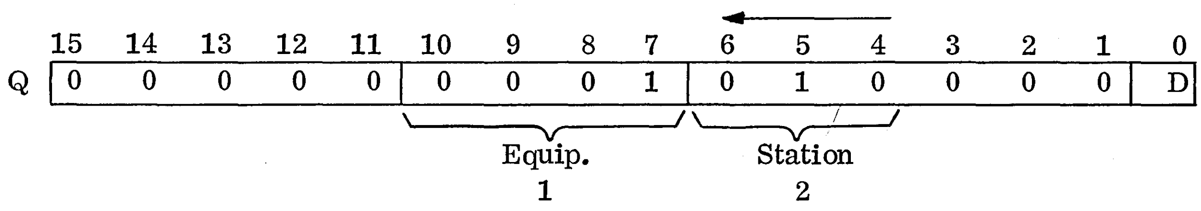
The grouping of the teletype, paper tape reader and paper tape punch is referred to as the low-speed package. The low-speed package is always equipment number 1. The various peripherals attached are referenced specifically with the S portion of the Q register, bits 4-6.

Peripheral	"S"tation
Teletypewriter	1
Paper tape reader	2
Paper tape punch	4

The format of Q for each of the low-speed peripherals is as follows:

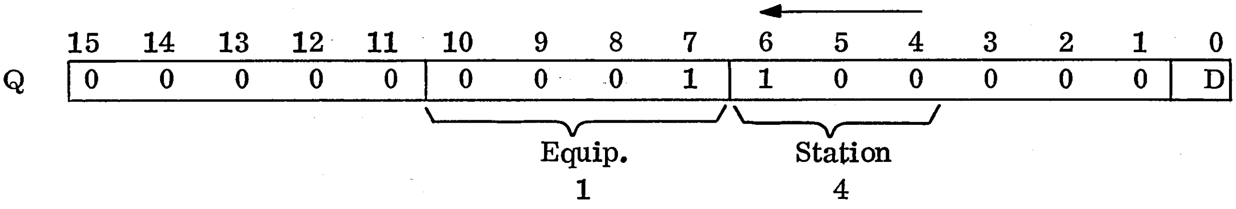


TELETYPEWRITER $0090/$0091



PAPER TAPE READER $00A0/$00A1

7.1.4



	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Q	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	D

PAPER TAPE PUNCH $00C0/$00C1

The programmer must load the Q register with the correct equipment, station and director setting prior to executing the desired I/O instruction.

The coding necessary to reference each of the low-speed peripherals for data is as follows:

```
LDQ       =N$0090     TTY FOR DATA
NOP
INP       -1          DATA IN A

LDQ       =N$0090     TTY FOR DATA
LDA       DATA        DATA IN A
NOP
OUT       -1          SEND DATA TO TTY

LDQ       =N$00A0     PTR DATA
NOP
INP       -1          DATA IN A

LDQ       =N$00C0     PTP DATA
LDA       DATA        DATA IN A
NOP
OUT       -1          DATA TO PTP
```

7.1.5 Reply or Reject

Control will be returned to the program after the execution of an I/O instruction at one of three locations: P+1, P+Δ, P+1+Δ. Control will be returned to P+1 when the controller accepts the command, NORMAL REPLY. Control will be returned to P+1+Δ when the controller rejects the command, EXTERNAL REJECT. If the controller fails to reply or reject within 6 microseconds, an INTERNAL REJECT is generated and control continues at P+Δ. The NOP instruction is inserted within the above coding to allow for an INTERNAL REJECT.

RESPONSE	ADDRESS	INSTRUCTION	Δ
INTERNAL REJECT →	2000	NOP	
EXTERNAL REJECT →	(P) 2001	INP	-1
NORMAL REPLY →	2002	Continue	

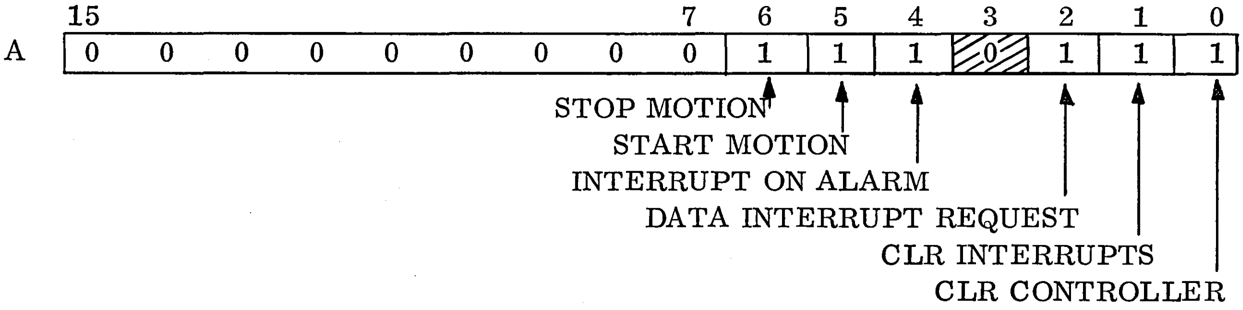
INTERNAL REJECT	P+ Δ	2001+(-1)=2000
EXTERNAL REJECT	P+1+ Δ	2001+1+(-1)=2001
NORMAL REPLY	P+1	2001+1=2002

7.1.6 Functions

I/O programming for the 1700 peripherals requires the programmer to connect with the desired peripheral by setting the Q register and issuing a function.

```
LDQ     =N$00A1     PTR FOR FUNC OR STATUS
LDA     FUNC        FUNCTION IN A
NOP
OUT     -1          SEND FUNCTION TO PTR
```

A function is a command or a group of commands sent to the controller. The function sent will vary according to the equipment. The paper tape reader allows the programmer to clear the controller, clear interrupts, select interrupt on data and/or alarm, start motion and stop motion. Each function corresponds to a bit in the A register. The function is requested if a 1 appears in the corresponding bit.



The above functions may be sent together, with the exception of the clear controller and clear interrupt; these must be sent separately.

7.1.6.1 Paper Tape Reader Example

The programmer may clear the controller to clear all logic and interrupts previously selected by loading Q with $00A1, by setting bit 0 of the A register and by issuing an OUT instruction.

```
LDQ         =N$00A1       PTR FUNC OR STATUS
ENA         1             CLR CONTROLLER
OUT         -1            FUNC TO PTR
```

The programmer may then start the motor on the paper tape reader. The start motor command causes the reader to begin moving paper tape and start reading.

```
LDQ         =N$00A1       PTR FUNC OR STATUS
ENA         $20           START MOTION
OUT         -1            FUNC TO PTR
```

The next step is to bring the data into the A register by setting the director bit of Q to 0 and issuing an INP instruction.

```
LDQ         =N$00A0       PTR FOR DATA
NOP
INP         -1            DATA INTO A
```

The program will continue to loop on the INP instruction until data has been read into the holding register of the paper tape reader. Once data is available it will be brought into the lower 8 bits of the A register. Note: the number of bits composing a data word will vary among the peripherals. The first frame will be in A and may be shifted to the upper 8 bits. Input will then be requested again for the lower 8 bits.

```
ALS         8             FIRST FRAME UPPER 8 BITS A
NOP
INP         -1            NEXT FRAME BROUGHT TO A
```

The entire 16-bit word is now in the A register and should be stored in the buffer area. Once the word is stored the program continues to bring data into the A register. A check should be made to determine when all requested words have been read from the reader.

```
           NAM    PTR
START      LDQ    =N$00A1    PTR FOR FUNC/STATUS
           ENA    1          CLR CONTROLLER
           OUT    -1         FUNC TO PTR
           ENA    $20        START MOTOR
           OUT    -1         FUNC TO PTR
           INQ    -1         PTR FOR DATA
           NOP
DATA       INP    -1         FRAME IN A
           ALS    8          DATA UPPER 8 BITS
           NOP
           INP    -1         16-BIT WORD IN A
           STA*   BUF        SAVE IN MEMORY
           LDA    WDCK       WORD CHECK IN A
           SAP    COMP       WHEN POSITIVE COMPLETE
           RAO*   WDCK
           JMP*   DATA       CONTINUE READING
COMP       SLS    0          STOP WHEN COMPLETE
WDCK       NUM    FFF0       CHECK FOR 16 WORDS
           END    START
```

The programmer may check for leader on the paper tape as has been done in the example on the next page.

7.1.6.1

1721 PTR - AUG '68
USDA

*Clear controller from console; cannot start motion and clear controller in same function.

```
                               NAM     BOOTSTRAP
                               ENT     START
P0000  E000     START          LDQ     =N$A1        PTR  DIR  FUNC
P0001  00A1
P0002  0A20                    ENA     $20          START MOTION*
P0003  03FE                    OUT     -1
P0004  0DFE                    INQ     -1           SET TO READ
P0005  0B00                    NOP
P0006  02FE     LOAD1          INP     -1           INPUT LEADER
P0007  0113                    SAN     3
P0008  18FD                    JMP*    LOAD1
P0009  0B00                    NOP
P000A  02FE     LOAD2          INP     -1           INPUT FRAME
P000B  0FC8                    ALS     8            SHIFT TO PACK
P000C  0B00                    NOP
P000D  02FE                    INP     -1           INPUT NEXT FRAME
P000E  6C04                    STA*    (ADDRES)     STORE WORD
P000F  0103                    SAZ     EXIT-*-1     EXIT ON ZERO WORD
P0010  D802                    RAO*    ADDRES       UPDATE ADDRESS
P0011  18F8                    JMP*    LOAD2        GO GET NEXT WORD
P0012  0014 P   ADDRES         ADC     *+2          LOAD AT P0014
P0013  0000     EXIT           NUM     $0           ZERO FOR SLS
                               END     START

I       00FF   START    0000P  LOAD1    0006P  LOAD2    000AP  ADDRES    0012P
EXIT    0013P
```

BOOTST 1725 ↙ Where BOOTSTRAP loaded

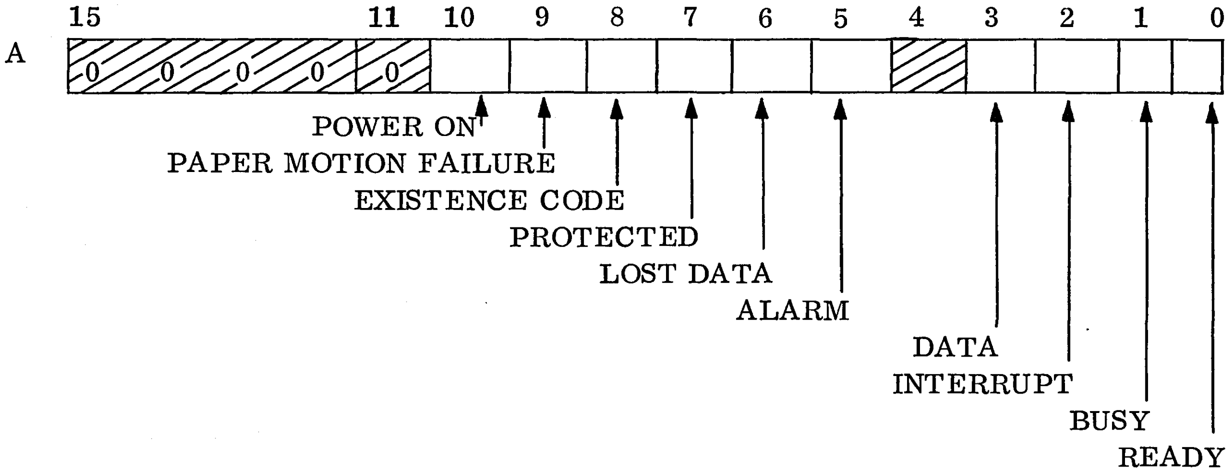
Set Stop switch up and program will stop after reading paper tape. Run and loaded PGM will execute.

7.1.7 Status

The paper tape reader has been programmed without the use of interrupts for each of the above examples. Status may be taken on the paper tape reader at any time to monitor the progress of the operation or to assure the program that the operation was completed correctly. Status is taken by setting the Q register and issuing an INP instruction.

```
LDQ       =N$00A1       PTR FUNC/STATUS
NOP
INP       -1            STATUS TO A
```

The status is now in the A register. The status conditions exist if a 1 is present in the corresponding bit position.



Ready (bit 0): Power is on and paper tape has been loaded into the reader. The preparations have been made known to the logic by pressing the READY switch on the paper tape reader console. The reader becomes not ready if a paper motion failure occurs or if the power is turned off.

Busy (bit 1): The paper tape reader is busy if a start motion command has been issued and no stop motion command has followed. Motion stops on a stop motion command, a paper motion failure, or if the power is turned off.

Interrupt (bit 2): An interrupt condition exists. Other status bits must be examined to determine the condition causing this interrupt.

Data (bit 3): The data hold register in the paper tape reader contains an 8-bit frame of data which is ready for transfer to the computer. Start motion must be set to receive this status. The status drops when the data hold register is emptied by transfer to the computer.

Alarm (bit 5): At least one of the following conditions exists in the paper tape reader: (1) paper motion failure (bit A9), (2) lost data (bit 6), or (3) power off (bit A10 is 0).

Lost data (bit 6): When in interrupt on data mode, paper motion continues after the data hold register is full. If the data is not transferred to the computer before the next frame appears, a lost data status occurs to show a frame has been passed. The time between frames is 2.857 milliseconds. The status drops when a clear controller command is sent. Lost data stops tape motion.

Protected (bit 7): The PROGRAM PROTECT switch is on. This switch on the paper tape reader works in conjunction with the PROGRAM PROTECT switch on the computer. If the switch on the computer is off and the PROGRAM PROTECT switch of the peripheral device is on, no action is taken but the status bit is set to indicate the switch is on. If the switch on the computer is set, all rules of program protection apply. The paper tape reader in this condition only accepts protected instructions.

Existence code (bit 8): The paper tape reader is attached. If the bit is a 1, the reader is missing from the particular computer system.

Paper motion failure (bit 9): No change in the feed hole circuit has occurred for 40 milliseconds while trying to read. The paper motion failure causes the reader to become not ready; it can only be made ready by pushing the READY switch or by a clear controller command. It is considered an illegal operation to send any other function code or a read command to the reader until the READY switch has been pressed or a clear controller has been issued.

Power on (bit 10): Power to the reader is on. If this bit is a 0, power is off.

7.1.8 Interrupts

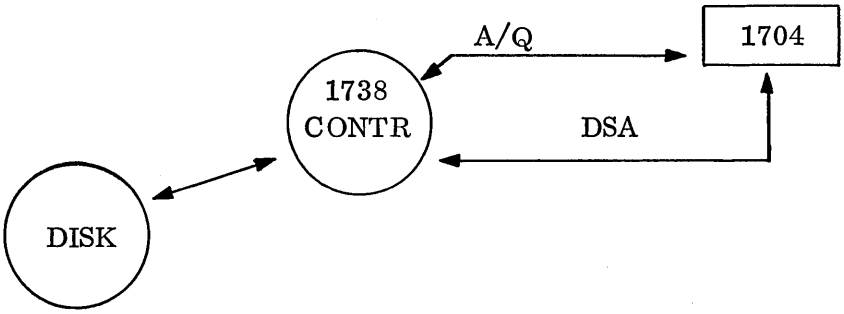
The paper tape reader may be programmed with interrupts by simply selecting the desired interrupts and exiting to the operating system or continuing execution of instructions within the program. When a selected interrupt is generated, control will be returned to the program.

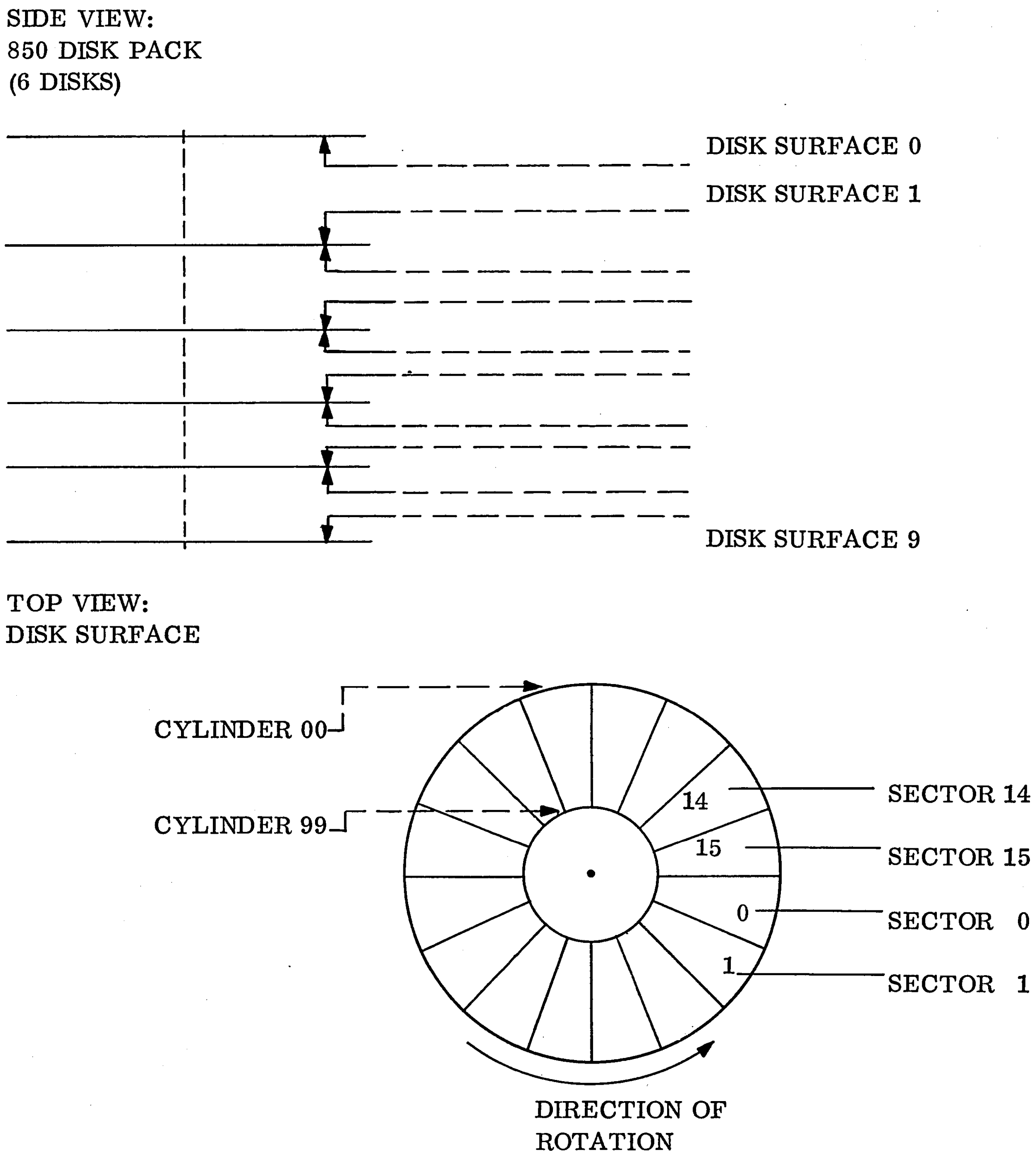
```
LDQ       =N$00A1     PTR FUNC/STATUS
ENA       1           CLR CONTROLLER
OUT       -1          FUNC TO PTR
ENA       $34         START MOTOR, ALARM, DATA
OUT       -1          FUNC TO A
Exit
```

When the interrupt returns control to the program, status must be taken to determine which of the two interrupts was generated, data or alarm. If the data interrupt was generated, the programmer brings the data into A and saves it. Once the data is saved, a check should be made to determine if all data has been transferred. If the operation was not complete, the programmer should reselect interrupts and exit, waiting for the next interrupt.

7.2 BUFFERED I/O EXAMPLE

The A/Q channel prohibits the execution of other instructions while data is being transferred. The reason is obvious: the Q register and A register, which are used for I/O, are also the arithmetic registers. A direct storage access (DSA) channel may be connected to the 1704. The DSA transfers data directly to memory, bypassing the A and Q registers. Therefore, a program may be executing while data is being transferred. The direct storage of data is referred to as BUFFERING. The A/Q channel is used to send functions and receive status, but the data is transferred via the DSA. The disk is an example of a buffered device.





SIDE VIEW:
850 DISK PACK
(6 DISKS)

TOP VIEW:
DISK SURFACE

853 contains 100 cylinders; 854 contains 200 cylinders.

Figure 14. Side and Top Views of 850 Disk Pack

7.2.1 Disk Functions

The program selects the disk by setting the Q register with the equipment number and the desired director bits. Throughout this discussion the disk shall be considered equipment number 8.

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Q	0	0	0	0	0	1	0	0	0	0	0	0	0		D	

EQUIP 8

The setting of the director bits will define the operation and the information to be sent from or received in the A register.

A	(A) depends upon D field of Q

Q		D

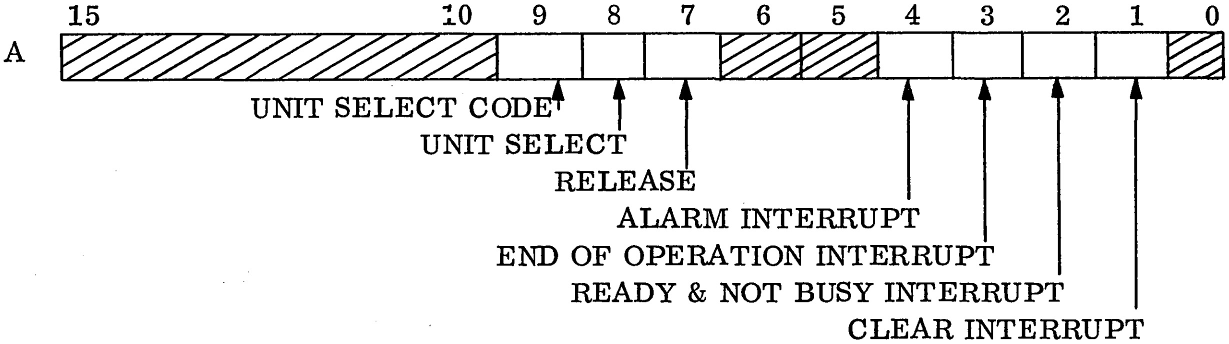
DISK FUNCTION CODES

Value Set in Q (Bits 02-00)	Output from A	Input to A
001	Director function	Director status
010	Load address	Address register status
011	Write	
100	Read	
101	Compare	
110	Checkword check	
111	Write address	

7.2.1.1 Director Function

D = 001 , OUT

```
LDQ       =N$0401     DISK FOR FUNC
LDA*      FUNC        FUNC IN A
NOP
OUT       -1          FUNC TO CONTROLLER
```



The clear interrupt function will clear all selected interrupts, allowing the programmer to select the interrupts he desires. Three interrupts may be selected: next ready and not busy status, end of operation, and alarm. The next ready and not busy interrupt occurs when the 1738 becomes not busy but still holds its ready status.

The end of operation interrupt allows the controller to inform the 1700 when it has completed an operation such as a data transfer. The alarm interrupt will notify the 1700 that an alarm condition has arisen. There are eight possible alarm conditions: not ready, checkword error, lost data, seek error, address error, defective track, storage parity error, and protect fault.

The release function allows an unprotected program to use the disk even though the protect switch on the disk is still set. A protected program must issue the release function. The next time a protected program accesses the disk, the disk will become protected and must again be released before it will become accessible to an unprotected program.

The unit select and unit select code will always be zero unless two disks are connected to the controller. Bit 8 is the unit select bit. It informs the controller that the program will select unit 0 or unit 1. Bit 9 indicates which unit bit 8 wishes to select. If 9 bit is a 0, unit 0 is selected; if it is a 1, unit 1 is selected. The controller ignores bit 9 unless bit 8 is set.

7.2.1.2 Load Address Function

D = 010 , OUT

Once the functions have been sent to the controller, the program notifies the controller of the beginning address on the disk to be used by the program. The Q register is loaded with the D portion set to 010 and the disk address (sector record address) is placed in the A register (Figure 14).

```
LDQ      =N$0402     DISK FOR DISK ADDR
LDA      =XDISKAD    ADDR IN A
NOP
OUT      -1          ADDR TO THE DISK
```

The controller will position the Read/Write heads on the requested address. The heads will be moved directly to the address forward or backward, depending on the current location of the Read/Write heads. The address held in the A register will be in the following format.

	15 8	7 4	3 0
A	CYLINDER	HEAD	SECTOR

The disk has been functioned and given the desired disk address. The next step will be to initiate one of three operations.

WRITE	D = 011
READ	D = 100
COMPARE	D = 101

7.2.1.3 Write Function

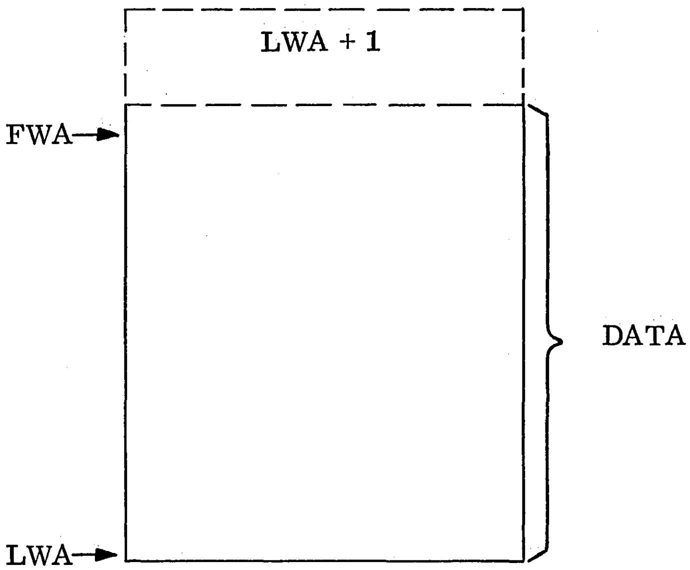
D = 011 , OUT

The write function code requests the controller to prepare to read data from memory and write it on the disk.

```
LDQ      =N$0403     DISK TO WRITE
LDA      =XMEMADR    MEMORY ADDR IN A
OUT      -1          WRITE OPERATION INITIATED
```

The controller expects to find the first word address minus 1 (FWA-1) of the buffer area in the A register when the write function is received. The controller goes into memory via the DSA to the FWA-1, at which location the controller extracts the last word address plus 1 (LWA+1). The controller keeps the LWA+1 and updates the FWA-1 until the two are equal; at this point the write operation is complete. The controller updates the address each time a 16-bit data word is transferred from memory.

DATA BUFFER FOR DISK



FWA - 1 MUST CONTAIN LWA + 1 OF BUFFER

Prior to issuing the write operation, the interrupts must be selected, the sector record address must be sent to the controller and the LWA+1 of the buffer area must be at the FWA-1.

```
LDQ       =N$0401      DISK FOR FUNC
LDA*      FUNC         SELECTED INT IN A
NOP
OUT       -1           INT SELECTED
LDQ       =N$0402      DISK FOR DISK ADDR
LDA       =XDISKAD     ADDR IN A
NOP
OUT       -1           HEADS POSITIONED
LDA       =XLWAP1      LWA+1 IN A
STA       FWAMI        LWA+1 AT FWA-1
LDQ       =N$0403      DISK TO WRITE
LDA       =XFWAM1      FWA-1 IN A
NOP
OUT       -1           OPERATION INITIATED
```

The 1704 continues executing instructions while the disk transfers data. When the data has all been transferred or an alarm condition has arisen, the 1704 will be notified.

7.2.1.4 Read Function

D = 100 ; OUT

The read function code follows the same programming procedure as the write function, the differences being the D setting of Q and the fact that the disk reads data into memory rather than writing data on the disk.

7.2.1.5 Compare Function

D = 101 ; OUT

The compare function code follows the same programming procedure as the read and write function codes. The compare function causes the controller to read data from the computer's memory and compare it with the data stored on the disk. If at any time during the compare, one word does not compare, the no compare status bit will be set. This function provides an extra check on the validity of the data transferred.

The checkword check (D=110) and write address (D=111) functions are used by the customer engineers.

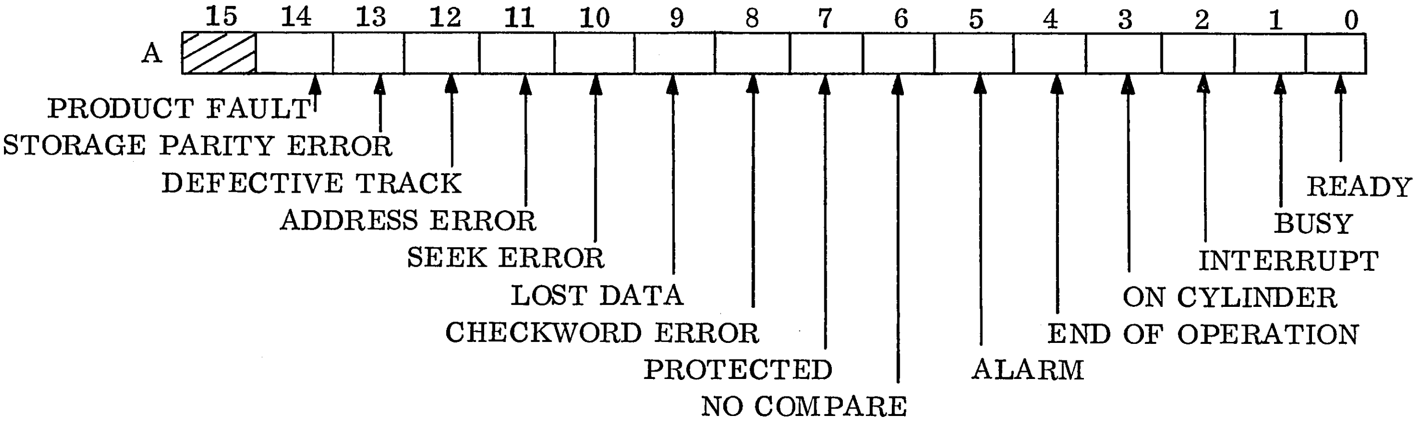
7.2.2 Disk Status

7.2.2.1 Director Status

D = 001 , INP

Status may be taken to monitor the operation of the disk. Once the disk generates an interrupt, status must be taken to determine which interrupt was generated. This is accomplished by loading the Q register with the D portion set to 001 and by issuing an INP instruction.

```
LDQ     =N$0401     DISK FOR STATUS
NOP
INP     -1          STATUS IN A
```



The ready status indicates that the unit is available. The busy bit indicates that the controller and/or the drive unit is presently involved in the performance of an operation. This bit is set with the acceptance of a load address, write, read, compare, checkword check, or write address function. At the completion of the function which set the busy status, the status will be cleared and the disk will become not busy. Once the disk is not busy, a new function may be issued.

The interrupt bit acknowledges that an interrupt has occurred. Further examination of A will determine which of the three selected interrupts was generated: bit 4 (EOP) and bit 5 (ALARM).

If neither bit 4 nor bit 5 is set, the programmer should check bits 0 and 1 for ready and not busy. If the alarm bit is set, the programmer must determine which of the eight alarm conditions caused the interrupt.

The on cylinder status bit 3 is set when the Read/Write heads have reached the sector record address initially sent to the controller via the A/Q channel.

7.2.2.2 Address Register Status

D = 010 ; INP

The programmer may request the disk to return the current position of the Read/Write heads at any time by selecting the disk as above but issuing an INP instruction.

```
LDQ         =N$0402       DISK FOR DISK ADDR
NOP
INP         -1            CURRENT ADDR IN A
```

The address will be in the same format used to send the address to the controller.

7.2.3 Summary, Buffered I/O

The DSA provides the 1704 with a means of storing data directly in memory; this permits the execution of instructions while transferring data. The program sends the function word, the sector record address, and stores the LWA+1 of the buffer area at the FWA-1 prior to initiating an operation. The operation is indicated by the D portion of Q, with the A register containing the FWA-1 of the buffer area. The controller interrupts the 1704 when a selected interrupt condition arises. The program takes status to determine which of the selected interrupts was generated.

CHAPTER VIII

SYSTEM REQUESTS

CHAPTER VIII - System Requests

CHAPTER VIII - System Requests (Cont)

Chapter VIII - System Requests (Cont)

8.1 OPERATING SYSTEMS

The 1700 computer system has two commonly used operating systems: the utility system and the mass storage operating system (MSOS). MSOS is a disk or drum oriented system and allocates the resources of the computer according to a priority system. The utility system is a much smaller system. It provides for assembly, loading, and execution of programs in a batch mode.

8.1.1 Utility

The 1700 utility system provides the 1700 computer with a means of loading and executing programs in configurations smaller than the minimum required for the 1700 operating system. The utility system requires 8K of core but no disk or drum. I/O is by way of the paper tape reader and paper tape punch, and listing and operator control is through the teletype.* Execution of jobs through the utility system can make use of the standard drivers provided by utility. The standard drivers provided are for the teletype, paper tape input, and paper tape output. It is possible, then, to operate these devices in one's own program by simply using a standard calling sequence.

8.1.2 MSOS

Under MSOS core is divided into two areas: foreground and background. The foreground is for system and process programs. System programs are those programs that make up the operating system, such as the job processor and drivers. Process programs are those application programs that are most important to the particular installation. For example, if the system is controlling a chemical plant operation, those programs monitoring the chemical process are the process programs. The process programs and system programs usually have the highest priority and have access to the resources of the computer first. Foreground is protected; this means the on-line process programs cannot be destroyed by programs in the background and they cannot be inadvertently brought into execution by background programs. However, background programs may make use of protected routines such as I/O drivers.

Background programs are run in a batch mode (serially) and run at the lowest priorities in the system. Programs in the background are called jobs. Assembling, compiling, and loading are examples of such jobs.

8.2 REQUEST PROCESSING

Three basic functions of the operating system are to: (1) allocate core space to those programs that want to use it, (2) communicate with the outside world, i.e., supervise I/O operations, and (3) allocate CPU time between the various programs. When a program wants one of these functions done, a request is made to the operating system.

*There are other options available.

8.2

A request takes the form of a transfer of control to the module of the operating system that processes requests (the request entry processor, entry point name MONI) followed by words containing the necessary parameters for the particular request.

The entry address for MONI is always located in core location F4 so every request is initiated by an indirect return jump through F4. The return jump will provide the linkage necessary. The parameter string length is different for different requests:

```
54F4        RTJ   ($F4)      TO MONI
xxxx }      Request code
xxxx }      and other parameters
```

In the first parameter word, bit positions 9 through 14 will be the request code. This is for all requests.

MONI saves the registers of the requesting program in a special core area called volatile storage and gives control to a request processor denoted by the request code in the parameter list. This processor must return control to the request exit processor which returns control to the requesting program.

The 1700 operating system provides the user with up to 30 monitor requests; 20 are reserved for the operating system. However, they may be replaced by user-written processors when the system is initiated. The other 10 requests may be added at initialization by including in the resident load the necessary programs with the required entry points. The number of possible request processors can be extended from 30 to 63 by reassembly.

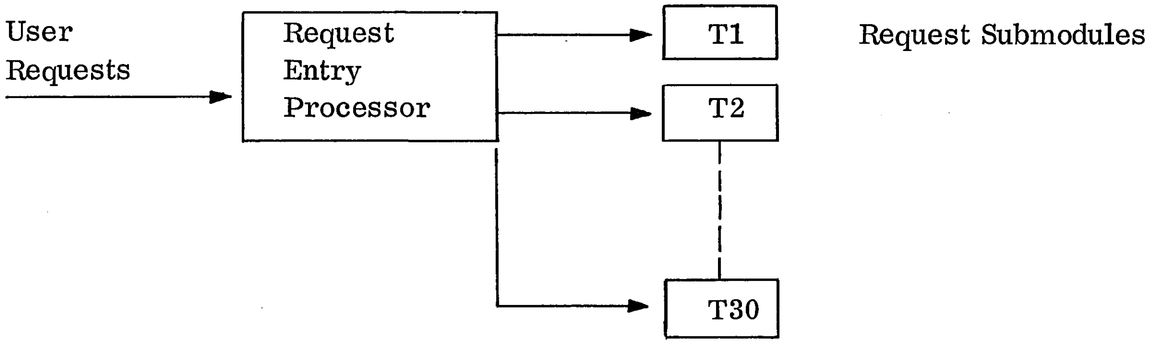
Each request processor is a separate submodule and has a coded entry point with one of the following names:

```
T1
•  \
•   >---------- reserved for system use
•  /
T20
T21
•  \
•   >---------- available to users
•  /
T30
```

The numerical part of each name is the request code. It corresponds to the value of an index to a table of request processor addresses contained in MONI.

MONI has these entry points as externals providing linkage with each appropriate request processor.

Since the numeric part of the entry point name for each request processor must correspond to its request code, a request code of 5 will provide entry through MONI to module T5.



Users can assemble an added request processor, assign a request code to it, and affix the entry point with T followed by the number for the request code and incorporate it as part of the system.

8.2.1 Summary of Request Processing

Note that all requests begin with the return jump to MONI and that MONI determines which type of request it is by examining the request code in the first word of the parameter string. Each type of request has a request processor to which MONI gives control, depending on the request code. When control is given to the request processor, several functions take place before control is returned to the requesting program.

It is important for the programmer to understand that a request only initiates action desired of the operating system. It does not do anything. In most cases, a request causes the desired action only to be put on a queue; control is returned immediately to the requestor at the next instruction beneath the parameter string.

For example, it is desired to write a message on logical unit 4:

```
REQ        RTJ-   ($F4)
           (Parameters for a
           write request on
           logical unit 4)
RET
```

Control is passed to MONI. MONI passes control to the Read/Write request processor which puts the request on a queue of other messages waiting to be printed on logical unit 4. Control then passes to the request exit routine which returns control to the requestor at RET. The message has not yet been written but it will be done in due time.

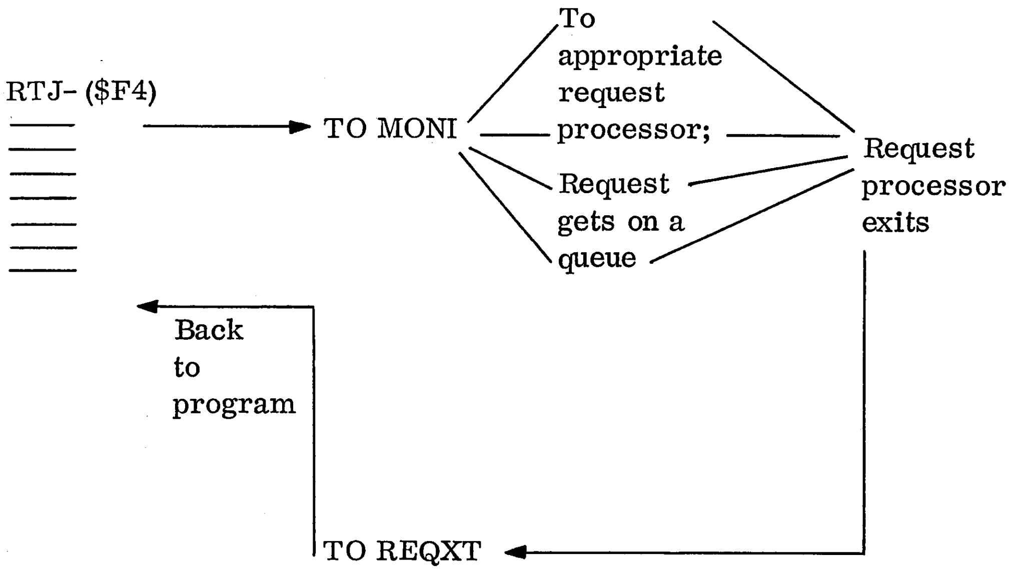


Figure 15. Flow of Requests

8.3 REQUESTS

The following requests are included in the standard operating system:

READ* WRITE* FREAD* FWRITE* SCHEDULE TIMER	Available to both foreground and background programs.
EXIT* CORE LOADER GTFILE STATUS*	Available only to background programs. Request processor modules for these requests must have the same residency as the job processor.
SPACE RELEASE	Available only to foreground programs.

INDIR is an indirect version of any of the listed requests.

The job processor can make <u>any</u> request.

*These requests are included in the utility system.

System macro calls are available to generate the code for the requests under the macro assembler, which runs under MSOS. Codes for the requests under the utility system must be coded by the programmer.

In this chapter those requests that are available to background programs will be discussed. Those requests available only to foreground programs will be discussed in Chapter 11.

Any of the allowable requests for background programs will be accepted by the operating system and put on the desired queue. They will not be rejected.

			RC	NUM
EXIT			5	0A00
FWRITE	ℓ, c, s, n, m, rp, cp, a, x*		6	0CXX
FREAD	ℓ, c, s, n, m, rp, cp, a, x*		4	08XX
WRITE	ℓ, c, s, n, m, rp, cp, a, x*		2	04XX
READ	ℓ, c, s, n, m, rp, cp, a, x*		1	02XX
SCHDLE	c, p, x		9	12XX
TIMER	c, p, x, Q, u	◄——Parameter in Q	8	10XX
CORE		◄———————Parameters in A and Q	11	1600
LOADER		◄———————Parameters in A and Q	7	0E00
GTFILE	c, f, s, w1, w2, x, rp, cp		13	1A01
STATUS	ℓ, ap, a, x		3	0600
INDIR	(ap)		-	-

General Definitions

ℓ	logical unit
c	completion address
s	starting address of data block
n	number of words
m	mode
rp	request priority
cp	completion priority
ap	address of parameter list
p	priority
u	units
f	index
x	mode of addressing indicator for c, s, n, f, and ap in STATUS
a	mode of addressing indicator for logical unit

Figure 16. Macro Calls for Requests Available to Background Programs

*For mass storage devices the programmer sets up two words following the request to contain the mass storage address.

8.3.1 EXIT Request - Request Code 5

The EXIT request is available only to background programs.

When control is to be given back to the job processor, the EXIT request is used. This may be when the program has reached a point where it has completely finished its goal and wants to terminate itself or when it wants to give up control while waiting for an I/O operation to take place.

The EXIT assembles into two words in the following manner:

$$\boxed{0}\underbrace{\boxed{000101}}_{\text{RC = 5}}\boxed{000000000}$$

```
        54F4
        0A00
```

macro call:

```
        EXIT
```

coded call:

```
        RTJ-        ($F4)
        NUM         $0A00
```

Most of the examples in this chapter have at least one EXIT request; check them for its use and assembly.

If this request were executed in a completion routine, normal processing of a job would resume at the location where it was interrupted. If this request was not executed in a completion routine, the job is considered complete.

The EXIT request simply causes a jump to the dispatcher which could be accomplished by:

```
        EQU         ADISP($EA)
        JMP-        (ADISP)
```

This would be a better way to code exits since it saves time and since protected programs cannot make EXIT requests.

In the background, the jump to the dispatcher can be used only under MSOS 2.0; under 1.0 it can only be used in the foreground.

8.3.2 Read/Write Requests - Request Code 1,2,4,6

For peripheral equipment that more than one program may use, a standard driver for each is written and incorporated into the operating system. The programmer makes use of the driver when he makes a Read/Write request to the operating system. Read/Write requests are available to both foreground and background programs.

When the programmer wants to communicate with a peripheral device, he makes a READ, WRITE, FREAD, or FWRITE request. Reads are used when information is to be brought into core and writes when information is to be transferred from core to the device.

When a Read/Write request is made, the transfer of data is only initiated. If the driver is busy working on another request, the new request is placed in a list of requests waiting to use the device. The position of the request on the queue will depend on the request priority. The request itself is not placed on a stack but is simply threaded. When the device is free and the transfer begins, there will be a period of time lapse before the complete transfer has taken place. During this time, control is given back to the requesting program so that it may execute any portion of the program that is not dependent upon the data. Therefore, when control returns to the instruction immediately following the parameter list, the programmer cannot assume that the transfer of data has been completed or, for that matter, even begun.

The programmer specifies a completion address in the request. Control will be given to this address when the request has been completed, according to the completion priority in the request.

This completion address is where the device driver will schedule reentry after the particular request has been completed. This scheduling of the completion address is done by the driver using the SCHDLE schedule request; this completion address is scheduled by priority. There are, then, parameters in the Read/Write requests to signify the priority of the request itself and the priority of the completion address.

8.3.2.1 Format of the Read/Write Request

REQ RTJ- ($F4)

<table>
<tr><td rowspan="8">Parameter List</td><td>0</td><td>15</td><td colspan="5">14 9
RC
1, 2, 4, 6</td><td>8
X
0, 1</td><td>7 4
RP
0 - 15</td><td>3 0
CP
0 - 15</td></tr>
<tr><td>1</td><td>15</td><td colspan="8">14 0
c
COMPLETION ADDRESS</td></tr>
<tr><td>2</td><td colspan="9">THREAD WORD = 0</td></tr>
<tr><td>3</td><td>15
ERROR
v</td><td>14
SHORT
READ</td><td>13
DEVICE
FAILED</td><td>12
M
0,1</td><td>11 10
A
0 0
0 1
1 0</td><td colspan="4">9 0
LOGICAL UNIT</td></tr>
<tr><td>4</td><td>15
0,1</td><td colspan="8">14 0
n
NUMBER OF WORDS</td></tr>
<tr><td>5</td><td>15
0,1</td><td colspan="8">14 0
s
FIRST WORD ADDRESS OF BUFFER</td></tr>
<tr><td>6</td><td colspan="9">15 0
msb
MOST SIGNIFICANT BITS OF MS ADDR.</td></tr>
<tr><td>7</td><td>15
0</td><td colspan="8">14 0
lsb
LEAST SIGNIFICANT BITS OF MS ADDR.</td></tr>
</table>

rc request code, 6 bits: WRITE = 2, READ = 1, FWRITE = 6, FREAD = 4

x relative/indirect indicator: 0 or 1

rp request priority, 4 bits, 0-15: position of request on driver's thread

cp completion priority: 0-15

c completion address, bit 15 set if (c)

thread word, is used by the system: equals zero when request is not active, nonzero when active

v error code passed to the completion address in Q and set in the parameter list when the request is completed

m mode bit: 0 = binary, 1 = ASCII

ℓ logical unit, 10 bits; modified by a

a settings for the a parameter (logical unit address indicator), 2 bits:

a = blank =00 : ℓ is a logical unit number
a = R = 01: ℓ is a signed increment (±1FF)
a = I = 10: ℓ is a core address (0<ℓ≤3FF)

s starting word of storage block, bit 15 set if (s)

msb-lsb set up by the programmer if logical unit specifies a mass storage device

The type of addressing used is determined by the request code. Background programs can access scratch only; sector 0, word 0 refers to the first word of scratch. These two words should be included in the request only if a mass storage device is involved. The macro call does not assemble them so the code must be added for them.

8.3.2.1

The macro call is:

```
                    FREAD
        REQ         READ        ℓ,c,s,n,m,rp,cp,a,x
                    FWRITE
                    WRITE
```

REQ will be on the 54F4, which will be generated. REQ+1 is the first word of the parameter string. The parameter list is the same for all four of the Read/Write requests.

An example of a coded call would be:

```
RTJ-        ($F4)
NUM         $0C01
ADC         COMPL
NUM         0
NUM         $1004
NUM         35
ADC         BUF
```

8.3.2.2 Request Code

There are four input/output requests handled by the Read/Write request processor RW, which has entry points T1, T2, T4, T6 for them.

	Request Code	NUM
READ	1	$02XX
FREAD	4	$08XX
WRITE	2	$04XX
FWRITE	6	$0CXX

The manner in which the data is to appear on the device is determined by which form of the request is used. FWRITE and FREAD are used for formatted transfers and READ and WRITE are used for unformatted transfers.

8.3.2.2.1 Format of Records

The format in which data may appear on a device is different for each device and driver. This information is given in the manual for each driver. The record formats for devices included in the minimum equipment configuration are outlined on the following page.

8.3.2.2.1.1 Teletype

FREAD

A format record for a teletype read operation consists of any number of characters followed by a carriage return. Format reading continues until a carriage return is detected or the requested number of words is transferred. If the requested number of words is transferred before the carriage return is detected, reading continues but no more data is transferred. A delete character will terminate data transfer until the next carriage return. The request will then be repeated. Line feed characters are not transferred in a format read. Before a format read on the teletypewriter begins, the break light is turned on; therefore, type in must be preceded by pressing the break release.

FWRITE

A format record for a teletype write operation consists of any number of characters following a carriage return and line feed character. A format write is the same as an ordinary write except that the driver supplies a carriage return and line feed character before beginning the transfer of data for a format write request.

READ

In an unformatted read, the number of words requested is filled starting at the specified address. Two characters fill one word; the first character is put into the upper half of the word. All carriage control must be provided by the programmer.

WRITE

In an unformatted write, two characters are transferred per word. The information must be stored in ASCII since no conversion is made from binary.

8.3.2.2.1.2 Paper Tape Reader and Paper Tape Punch

A format record for the paper tape reader and punch depends on mode. In ASCII mode, it is any number of characters preceding a carriage return. In binary mode, it is any number of words preceded by a word which represents the ones complement of the number of words to be transferred and followed by a word containing a checksum. If the number of words to be transferred is greater than 21,759, the first word represents the ones complement of the number of words to be transferred plus 256.

If the first character of the header word in a format record (requested in binary mode) is an asterisk, the record is read in ASCII mode. This allows ASCII and binary mode records to be mixed.

8.3.2.2.1.2

FREAD, ASCII

For a format read, ASCII mode, characters are read until a carriage return is detected or the requested number of words has been transferred. If the requested number of words is transferred before a carriage return is detected, reading will continue but no additional data will be transferred. Nulls (blank tape) preceding a format record are skipped; no data transfer takes place until a non-blank character is detected. Line feed characters detected during the read are not transferred; delete characters are ignored. All characters are checked for even parity.

FREAD, Binary

Format reading in binary mode continues until the word count is exhausted or the requested number of words has been transferred. If the latter occurs before the count is exhausted, reading will continue but no additional data will be transferred. The checksum is verified after the entire record is read. If the first character of the header word is an asterisk, the record is read in ASCII mode in spite of the binary mode declaration. Nulls (blank tape) preceding a format record are skipped.

FWRITE, WRITE

A format write is the same as an ordinary write request except the driver supplies carriage return and line feed characters after the end of the operation.

In ASCII mode format write, characters are transferred from the specified core block until the word count is exhausted. A parity bit is added to each character. In binary mode, format write, characters are transferred until the word count is exhausted. Word count is the first word on the tape and is the ones complement of block size. The driver supplies a checksum word after the data.

Checksum

The driver generates the checksum word on a paper tape by accumulating a sum of the word count word (which is the complement of the data block size and is output on the tape first) plus all the data words, disregarding any overflow. This sum is then complemented and output as the last word in the block on the tape. For example, if the data words are $F001, $E001, $E002, $E003, and $B005, the word count will be 5 and the sum of the data 4010. The checksum will be $BFF4 which is $FFFA (complement of 0005) plus $4010, which is $400B; the complement of $400B is $BFF4. The tape will contain FFFA, F001, E001, E002, E003, B005, BFF4.

When the checksum word is read on input, it will result in zero when the word count word is read, the data words added to it, and then the checksum word is added to that. For example, on the above tape the word count word FFFA is read first, the data totalling 4010 is added to it as the data is read, giving 400B. When the checksum BFF4 is read and added, the sum will be FFFF arithmetically, which will be 0000 by the subtractive adder.

The paper tape on page 8-14 contains the data used in these examples.

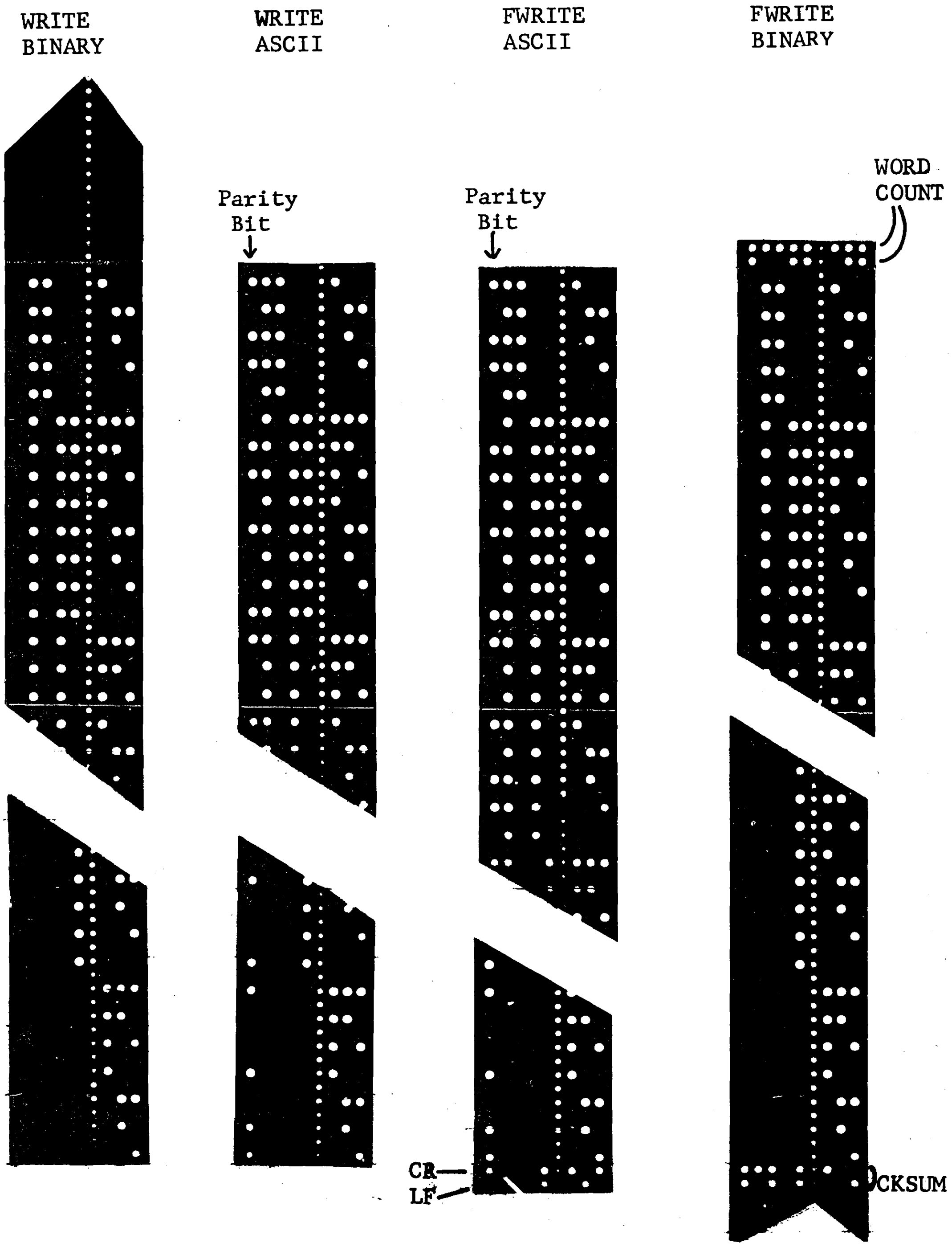
The following example program shows the paper tape output for all four kinds of writes. It writes 100 words of data from a buffer BUF. The data in the buffer is $1\text{-}64_{16}$ $(1\text{-}100_{10})$. The buffer is written out from the end to the beginning.

```
                       NAM       TEST WRITE INSTRUCTIONS
                       ENT       WRITE
P0000 0000    WRITE    0         0
              *                                    THIS IS HOW TO GENERATE DATA
              *                                    IN A PROGRAM FOR A TEST
P0001 0C63             ENQ       99
P0002 0A00             ENA       0
P0003 0901    DATA     INA       1
P0004 6A21             STA*      BUF,Q
P0005 0142             SQZ       PUNCH-*-1
P0006 0DFE             INQ       -1
P0007 18FB             JMP*      DATA
              *
              *                                             WRITE, BINARY
P0008 54F4    PUNCH    RTJ-      ($F4)
P0009 0401             NUM       $0401
P000A 0000             NUM       0
P000B 0000             NUM       0
P000C 0003             NUM       $0003
P000D 0064             NUM       100
P000E 0025 P           ADC       BUF
              *                                             WRITE, ASCII
P000F 54F4             RTJ-      ($F4)
P0010 0401             NUM       $0401
P0011 0000             NUM       0
P0012 0000             NUM       0
P0013 1003             NUM       $1003
P0014 0064             NUM       100
P0015 0025 P           ADC       BUF
              *                                             FWRITE, ASCII
P0016 54F4             RTJ-      ($F4)
P0017 0C01             NUM       $0C01
P0018 0000             NUM       0
P0019 0000             NUM       0
P001A 1003             NUM       $1003
P001B 0064             NUM       100
P001C 0025 P           ADC       BUF
              *                                             FWRITE, BINARY
P001D 54F4             RTJ-      ($F4)
P001E 0C01             NUM       $0C01
P001F 0000             NUM       0
P0020 0000             NUM       0
P0021 0003             NUM       $0003
P0022 0064             NUM       100
P0023 0025 P           ADC       BUF
              *
P0024 14EA             JMP-      ($EA)
P0025 0064    BUF      BZS       BUF(100)
                       END       WRITE
```

8.3.2.2.1.2

```
00FF   WRITE   0000P DATA    0003P PUNCH   0008P BUF    0025P
```

(NOTE THAT DATA IS NOT BLOCKED)



8.3.2.2.1.3 Mass Storage Addressing

When the logical unit in a read/write request specifies a mass storage device, the address to be accessed must be in the two words following the parameter list. This must be set up by the programmer. For example:

```
----
READ        -,-,-,-,-,-,-,-,-
NUM         $0000,0002              (sector 2)
----
```

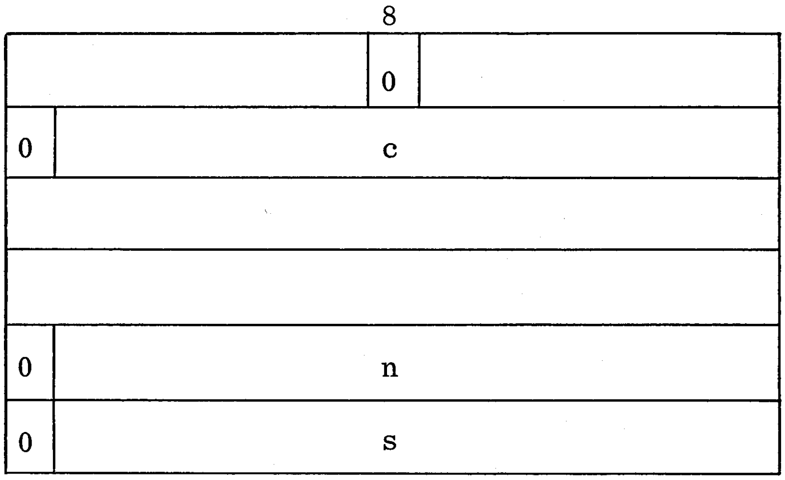
Since all transfers to and from mass storage do not have a mode or format associated with them; that is, what is written on the disk is a mirror image of the contents of core, the.formatted and unformatted forms of the request determine the type of addressing that will be used. Formatted requests mean sector addressing and Read/Write assumes word addressing. Check the manual for the driver for the exact details of addressing mass storage. Sector addressing can always be used; word addressing can only be used under systems which have the disk word driver.

In the background a program can access only the scratch area of the disk or drum through read/write requests. Therefore, sector one or word one in the background is sector or word one of the scratch area.

8.3.2.3 X Bit

The x bit is used in conjunction with the cp, n, and s parameters to indicate direct, indirect, or relative location of those parameters. It will be discussed as it relates to each parameter.

As a rule, the x bit = 0 is satisfactory for background programs, with all the associated parameters direct.



Its use as relative (for run anywhere programs) will be discussed in detail in Chapter 11. When the x bit = 1, the c and s parameters must be relative!

8.3.2.4 Request Priority

The request priority indicates to the operating system the priority of this request in relation to the other programs requesting use of the device. The request priority has nothing to do with the running priority of the program. In the background the rp parameter is always 0.

Since the request priority determines the position of the request on the queue for the driver, the reads and writes done by jobs at priority 0 will always be on the bottom of the queue and therefore will be done last.

This is why, for example, if a job and a process program are both writing messages on a remote teletype, the messages will be interspersed together with the process program's messages taking priority over the job's.

8.3.2.5 Completion Priority

The completion priority is the priority at which the completion routine is to run. For background jobs, cp is always 1. Since the running priority for jobs is always 0, the completion priority of 1 will cause the job to be interrupted, when the driver has finished the I/O, and the completion routine to be entered. (When the completion routine exits, control is returned by the dispatcher to the location where the job was interrupted.)

A completion priority greater than the running priority will always cause a pseudo interrupt of the program and will cause the priority to be changed higher and the completion address entered. A cp equal to or less than the running priority will cause the completion routine not to be entered until after the program exits and the priority structure works down to the desired level.

8.3.2.6 Completion Address

The completion address is scheduled by the driver as a program when the transfer is completed, or terminated by an error. The completion address is scheduled at a particular priority specified by the cp parameter. In the background cpis always 1.

This completion routine is generally short because, under the utility system, when it is entered the interrupt mask is set to prevent external interrupts from occurring. Under MSOS, interrupts are not locked out while the completion routine is being executed, so process equipment can interrupt.

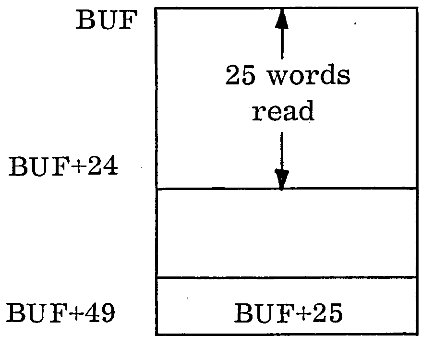
The completion address may be omitted; then no address is scheduled upon completion.

When control is given to the completion address, the Q register will contain a 3-bit code in the upper three bits of Q. These bits can also be found in the v field of the fourth word of the request.

Bit	Value	Condition
15	0	Error free operation
	1	Error occurred (device failure)
14	0	Requested number of words transferred
	1	Less than requested number of words transferred during a read or format read (short read)
13	0	Error occurred because device was not ready
	1	Error was due to failure of the device; device is ready

The Q register should always be checked for the possibility that an error occurred. For example, if a short read occurred on a READ request, bit 14 would equal 1. Bit 15 would indicate whether it was a legal short read (Q15=0), or due to device failure (Q15=1). If it was due to device failure, bit 13 would indicate whether the device is ready or not.

The programmer could find out how much of the data was transferred by the last word of the data buffer. This word would contain the address plus 1 of the last word transferred. Thus, by subtracting the contents of the last word in the buffer from the first word address of the buffer, the number of words can be found. If, for example, a read of 50 words was requested but only 25 words were read:



```
LDA  =XBUF          (A) = BUF
SUB  BUF+49             - BUF+25
                    (A) = 25
```

Each appropriate driver manual will indicate possible error conditions.

The completion address may be specified in several ways. The x parameter determines the mode of addressing that is used for the completion address.

x	c	The meaning of c
0 or blank	c	c is the completion address e.g., FWRITE -, COMPL, -, -, -, -, -, -, 0 COMPL is assembled as an absolute address (program relocatable).
≠0 ≠blank	c	c is relative c is a positive increment added to the address of the first word of the parameter list to locate the completion address. i.e., FWRITE -, COMPL-*+1, -, -, -, -, -, -,X In the example the completion address is at COMPL.
-	(c)	c in parentheses represents an index to the system library. x has no meaning.

Note that there is no indirect version for the completion address.

The option (c) cannot be used in the background because the programs on the system library may not be scheduled from the background; see Chapter 11.

A completion routine in one's own program may be scheduled in one of the following ways:

```
0001                             NAM   D - R/W EXAMPLE
0002                             ENT   GO
0003                     GO      FWRITE    4, COMPL1, EXAMPLE, 9, B, 0, 1, A, 0
0003   P0000  54F4                                ↖                            ↖
0003   P0001  0C01
0003   P0002  0009 P ◄────────── completion address
       P0003  0000
0003   P0004  0004
0003   P0005  0009
       P0006  001AP
0004                             EXIT
0004   P0007  54F4
0004   P0008  0A00
0005                     COMPL1

0006                             FWRITE    4, COMP2-*+1, EXAMPLE-*+5, 9, B, 0, 1, A, X
0006   P000B  54F4                                 ↖                             ↖
0006   P000C  0D01
0006   P000D  0008 ◄──────────── Relative distance to completion address
       P000E  0000
0006   P000F  0004
0006   P0010  0009
       P0011  000E
0007                             EXIT
0007   P0012  54F4
0007   P0013  0A00
0008   P0014  0172   COMP2       SQM   ERROR
```

Note that the first assembles with a P following the completion address, meaning that it is direct and will be absolutized at load time (the actual core address filled in); so the program will be relocatable.

The second example is relative to the first word of the parameter list. Also, in the second example, note that the 8 refers to the location where the parameter is but that the parameter address must be relative to the first word of the parameter list, hence COMP2-*+1. The x bit must be set in the first word.

8.3.2.7 Thread Word

The thread word must always be initialized to 0 or the request cannot be executed. The thread word is used by the operating system for the queue for requests and alsofor the completion. It is non-zero during the entire time the request is being processed and while the I/O is being done. It does not become zero again until the operation is complete.

8.3.2.8 Error Code

The 3-bit v field in the fourth word of the parameter string is the same 3-bit error code that comes back to the Q register at completion. It will be set by the driver whether or not there is a completion routine. Therefore, a program not using a completion routine may check the v field for errors.

8.3.2.9 Mode

The m parameter tells the driver whether the data is stored in core in Binary or ASCII. When m is A (or, if coded, 1) it means the data is stored in ASCII, and B (or, if coded, 0) means Binary.

8.3.2.10 A Field

The a bits modify the logical unit field, indicating whether the logical unit field contains the actual logical unit number, an indirect address containing the LUN, or a relative distance to LUN. The a bits will be discussed under the logical unit heading.

8.3.2.11 Logical Unit

The logical unit may be specified directly (i.e., ℓ=4), a word may be specified that contains the logical unit, or a relative distance to the address of the LUN may be specified. The a parameter indicates which form of the ℓ parameter the programmer has used.

a	Meaning of the ℓ parameter
= 0 or = A or = blank	the parameter is the logical unit e.g., FWRITE 13, -, -, -, -, -, -, 0, -
= 2 or = I	ℓ is indirect ℓ is an absolute location that contains the logical unit e.g., FREAD $FD, -, -, -, -, -, -, I, - is used when standard comments (input) wanted. ℓ cannot be larger than $3FF.
= 1 or = R	ℓ is relative ℓ is a number that specifies the number of words to a word that contains the logical unit. The ℓ parameter in this case can be positive or negative but must not exceed + or - $1FF. e.g., READ LUN-*+3, -, -, -, -, -, -, R, - where LUN contains the logical unit.

Note the restriction that when using the I option the absolute address must be within the range of 000 to 3FF which in most cases precludes the use of the I option when the location is within ones own program, except for system units.

The following are the core locations in the operating system which contain the logical unit numbers of the standard devices:

Device	Core Location
Input comment device	$FD
Output comment device	$FC
Standard print output device	$FB
Standard binary output device	$FA
Standard input device	$F9
Mass storage library	$C2
Mass storage scratch	$B3

Each piece of equipment has a logical unit number assigned to it for the system to use. Also, one piece of equipment may have more than one logical unit number, depending on the use. It is not to be confused with the equipment number dialed on the controller nor with the unit number set on the unit (i.e., tape unit). It could be thought of as the system number of the driver.

The logical unit numbers will be different at each installation but the following numbers are very commonly used:

2 - PTR

3 - PTP

4 - TTY

5 - CR

6 - MT #1

7 - MT #2

8 - DISK

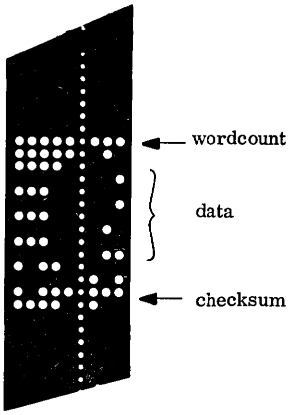
9 - LP, system driver

10 - LP, FORTRAN driver

If the programmer does not want to change the logical unit for this request, during his program he could specify it directly, as in the following example.

```
0001                         NAM    A R / W EXAMPLE A
0002                         ENT    START
0003        0003            EQU   PTP(03)
0004  P0000 F001   BUF       NUM    $F001, $E001, $E002, $E003, $B005, $C004
      P0001 E001
      P0002 E002
      P0003 E003
      P0004 B005
      P0005 C004
0005               START     FWRITE   PTP, XX, BUF, 5, B, 0, 1, , 0
0005  P0006 54F4
0005  P0007 0C01
0005  P0008 000F P
      P0009 0000
0005  P000A 0003 ←——————— logical unit
0005  P000B 0005
      P000C 0000 P
```

Output



Note that the assembler put 0003 in the logical unit position (P000A) of the code for the FWRITE. Note also that only 5 of the 6 words in the buffer were written out.

See also the logical unit specification in the macro calls in the completion address example in section 8.3.2.6. The A specified that the logical unit field (4) was absolute.

On the other hand, if the programmer wanted to use one of the standard devices, such as the comment device, he would specify an indirect address in the following form:

```
0004                     X1      FWRITE    $FC,XB,STATEM,17,A,0,1,I,0
0004   P0011  54F4                           ↖                      ↖
0004   P0012  0C01
0004   P0013  001A P
       P0014  0000
0004   P0015  18FC  ←——————  location containing LUN
0004   P0016  0011
       P0017  0000 P
```

The programmer may want to change the logical unit, depending on the conditions in his program; therefore, he may have a location within his program in which to store the logical unit number. The following example would be a way to code a program to write one message (from M21) on a LOG unit and the COMMENTS unit, then write another message (from M22) on the COMMENTS unit only. The entry point is GO TO, and LOG is EQU'd to the LOG unit while COMMENTS is EQU'd to $FC. Note that the word in the program which contains the logical unit number must be addressed relatively because its absolute address may not fit in the 10-bit ℓ field. Note also that the completion and buffer addresses may be addressed absolutely in this same macro call because the x bit is not set for a relative logical unit word.

```
0003  P0000  5448    M21     ALF     *,THIS MESSAGE APPEARS ON THE LOG*
      P0001  4953

0004  P0010  414E    M22     ALF     *,AND THIS ON THE COMMENTS DEVICE.*

0005  P0020  0001    LUNIT   BSS LUNIT
0006  P0021  0B00    X1      NOP
0007                         FWRITE   LUNIT-*+3, COMPLETE, M21, 16, A, 0, 1, R, 0
0007  P0022  54F4
0007  P0023  0C01
0007  P0024  002B P
      P0025  0000
0007  P0026  17FC    ◄————    relative distance to LUNIT
0007  P0027  0010
      P0028  0000 P
0008                         EXIT
0008  P0029  54F4
0008  P002A  0A00
0009  P002B  1CF5    COMPLETE JMP* (X1)
0010  P002C  C000    GOTO    LDA    =XLOG
      P002D  000B
0011  P002E  68F1            STA*   LUNIT
0012  P002F  58F1            RTJ*   X1
0013  P0030  017E            SQM    ERROR
0014  P0031  C0FC            LDA    COMMENTS
0015  P0032  68ED            STA*   LUNIT
0016  P0033  58ED            RTJ*   X1
0017  P0034  017A            SQM    ERROR
0018                         FWRITE   $FC, OK, M22, 17, A, 0, 1, I, 0
0018  P0035  54F4
0018  P0036  0C01
0018  P0037  003E P
      P0038  0000
0018  P0039  18FC
0018  P003A  0011
      P003B  0010 P
0019                         EXIT
0019  P003C  54F4
0019  P003D  0A00
0020  P003E  0170    OK      SQM    ERROR
```

8.3.2.11

What other information does the programmer need to supply the operating system to perform an I/O operation? The operating system needs to know how many words are to be transferred, where in core the information is stored for a write operation or, for a read, where it is to be stored. This is done by specifying the remaining parameters.

8.3.2.12 Number of Words

The n parameter is the number of words requested to be transferred. A read will transfer one record, so if less than the requested number of words is in the record (i.e., on paper tape, see 8.3.2.2.1.2) a short read will occur. This is perfectly legal since it allows the programmer to specify a maximum but get the actual number.

The location of the number of words can be specified in several ways:

x	n	Meaning of n
-	n	n is the length of the block to be transferred; x has no meaning. e.g., FWRITE -,-,-,6,-,-,-,-,- In this case 6 words would be transferred.
=0 =blank	(n)	n is indirect. It is a core location containing the block size. e.g., WRITE -,-,-,(n),-,-,-,-,0 n contains the block size.
≠0 ≠blank	(n)	n is relative. It is the relative distance to a core location containing the block size. e.g., READ -,-,-,(N-*+4),-,-,-,-,X N in this case contains the number of words to be transferred.

See the other read/write examples for illustrations of the n parameter.

8.3.2.13 Starting Address of Buffer

The location in core in which the data to be transferred is stored is indicated by the s parameter. The s refers to the first word of the data block. The buffer address can be specified in several ways, again in conjunction with the x bit.

x	s	Meaning
=0 =blank	s	s is the starting address of the data block in core. e.g., FWRITE -,-,BUF,-,-,-,-,-,0 where BUF is the starting address.
=0 =blank	(s)	s is indirect. It is the core location that contains the absolute address of the data block. e.g., READ -,-,(BUFADR),-,-,-,-,-,0 BUFADR contains the absolute address of the starting address.
≠0 ≠blank	s	s is relative. s is a positive increment added to the address of the first word of the parameter list to form the starting address. e.g., FREAD -,-,BUF-*+5,-,-,-,-,-,X BUF would be the first word of the data block.
≠0 ≠blank	(s)	s is double relative. s is a positive increment added to the address of the parameter list to form the address of a word that contains another relative indicator. This indicator is added to the first word of the parameter list to form the address of the data block. e.g., WRITE -,-,(BUFREL-*+5),-,-,-,-,-,X BUFREL contains a number to be added to the first word of the parameter list to locate the buffer.

The above specifications mean one can specify the address directly, relatively or indirectly. If the buffer were within one's program and if the request were always going to deal with this one buffer, one could reference in one of the following ways.

8.3.2.13

```
0001                               NAME - R/W EXAMPLE
0002  P0000  5448    STATEM        ALF   *,THIS IS AN EXAMPLE FOR S PARAMETER*
      P0001  4953
      P0002  2049
      P0003  5320
      P0004  414E
      P0005  2045
      P0006  5841
      P0007  4D50
      P0008  4C45
      P0009  2046
      P000A  4F52
      P000B  2053
      P000C  2050
      P000D  4152
      P000E  414D
      P000F  4554
      P0010  4552
0003                               ENT    X1
0004                 X1            FWRITE    $FC,XB,STATEM,17,A,0,1,I
0004  P0011  54F4
0004  P0012  0C01
0004  P0013  001A P
      P0014  0000
0004  P0015  18FC
0004  P0016  0011
      P0017  0000  P <------------ Starting Address
0005                               EXIT
0005  P0018  54F4
0005  P0019  0A00
0006  P001A  0179    XB            SQM    ER1

0007                               FWRITE   $FC,XC-*+1,STATEM-*+5,17,A,0,1,I,X
0007  P001B  54F4
0007  P001C  0D01
0007  P001D  0009
      P001E  0000
0007  P001F  18FC
0007  P0020  0011
      P0021  7FE3  <-------------- Starting Address, Relative
0008                               EXIT
0008  P0022  54F4
0008  P0023  0A00
0009  P0024  181C    ER1           JMP*    ERROR
```

The second example shows that even though the relative address must be in the positive direction the data block can come before the request in the program. This makes use of the wrap-around feature of the machine's 15-bit address arithmetic.

1C + 7FE3 = 7FFF, which is a zero in 15 bits, indicating P0000.

If, on the other hand, the request may be used to read or write a number of data tables, an indirect method might be used.

```
0010  P0025  C000    XC      LDA   =XSTATEM
      P0026  0000 P
0011  P0027  6818            STA*   PT
0012                         FWRITE  $FC,XD,(PT),17,A,0,1,I,0
                                              ↖            ↖
0012  P0028  54F4
0012  P0029  0C01
0012  P002A  0031 P
      P002B  0000
0012  P002C  18FC
0012  P002D  0011
      P002E  803F  <-------- Starting Address, Indirect
0013                         EXIT

0013  P002F  54F4
0013  P0030  0A00
0014  P0031   017C   XD      SQM   ER2

0015  P0032  C000            LDA   =XSTATEM-EX-1
      P0033  7FC9
0016  P0034  680B            STA*   PT
0017                 EX      FWRITE  $FC,EC-*+1,(PT-*+5),17,A,0,1,I,X
0017  P0035  54F4
0017  P0036  0D01
0017  P0037  0006
      P0038  0000
0017  P0039  18FC
0017  P003A  0011
      P003B  8009
0018                 EC      EXIT
0018  P003C  54F4
0018  P003D  0A00
0019  P003E  1802    ER2     JMP*   ERROR
0020  P003F  0001    PT      BSS   PT
```

8.3.2.14 Setting Up RW requests in Background

Jobs are supposed to run at priority 0 and their completion routines are supposed to run at priority 1. There are three ways the input/output requests can be set up (for jobs) to maintain control in the program.

1. Looping on the thread word

2. Looping on a flag set by the completion routine

3. Scheduling out of the completion routine

The example to use would be an input of a card buffer which would then be written on the teletype.

8.3.2.14.1 Looping on the Thread Word

```
              RTJ-        ($F4)
              NUM         $0801       FREAD
              NUM         0           NO COMPL
THR           NUM         0
              NUM         $1002       ASCII, PTR
              NUM         35          35 WORDS
              ADC         BUF

WAIT          LDA*        THR
              SAZ         WRITE-*-1
              JMP*        WAIT

WRITE         -           -
```

The thread word will be non-zero until the input is finished; then the buffer can be written out. This technique hangs the computer in a loop and locks out any lower priority operations. Of course, a job is running at 0 so this will not matter. However, this method should not ever be used in foreground programs.

8.3.2.14.2 Looping on a Flag

```
              RTJ-        ($F4)
              NUM         $0801       FREAD
              ADC         COMPL       COMPL
              NUM         0
              NUM         $1002       ASCII, PTR
              NUM         35          35 WORDS
              ADC         BUF
WAIT          LDA*        FLAG
              SAN         WRITE
              JMP*        WAIT
WRITE         ENA         0
              STA*        FLAG
               {
                 ≀
                 EXIT
      COMPL      SQM        ERR-*-1      ⎫
                 ENA        1            ⎪
                 STA*       FLAG         ⎪
                 EXIT                    ⎬   Closed Routine
      ERR        ≀                       ⎪
                 JMP*       COMPL+1      ⎪
      FLAG       BSS        FLAG(1)      ⎭
```

This method does the same thing as looping on the thread word, except it provides for a completion routine where errors can be checked. It should not be used in the foreground either because it locks out priorities lower than the completion priority.

8.3.2.14.3 Scheduling Out of the Completion Routine

This method may be used only under MSOS.

```
                 RTJ-       ($F4)
                 NUM        $0801        FREAD
                 ADC        COMPL        COMPL
                 NUM        0
                 NUM        $1002        ASCII, PTR
                 NUM        35           35 WORDS
                 ADC        BUF          BUFFER ADDR
                 EXIT
      COMPL      SQP        OK-*-1       CHECK ERRORS?
                 RTJ*       ERR
      OK         SCHDLE     WRITE, 0, 0  SCHEDULE WRITE
                 EXIT
      WRITE      ≀
```

This method provides for an exit immediately after the request is initiated at priority 0, to wait for completion. When the completion routine is entered at priority 1, a check is made for errors and then a schedule request is made for the address WRITE to be entered at priority 0, after the completion routine does its exit. (The schedule request is covered in the next section.)

Scheduling out of the completion routine is a good method to learn to use because it can be used by either background or foreground programs, it maintains a priority scheme if it is desired that the I/O completion routines run at a different priority from the main body of the program (either higher or lower), and it does not lock out lower priority operations.

Any coding which can be done in the program before the data is needed can be inserted after the ADC BUF and before the EXIT. That coding will run at priority 0, the priority of the main body of the program.

8.3.2.14.3

A method similar to this is often used; it works in the background but does not maintain any priority scheme:

```
          RTJ-      ($F4)
          NUM       $0801
          ADC       COMPRD
          NUM       0                 Initiate Read
          NUM       $1002
          NUM       35
          ADC       BUF
          EXIT
COMPRD    SQP       WRITE-*+1
          RTJ*      ERR
WRITE     RTJ-      ($F4)
          NUM       $0C01
          ADC       COMPWR            Initiate Write
          NUM       0
          NUM       $1004
          NUM       35
          ADC       BUF
          EXIT
COMPWR
```

In the above example, the read is initiated at priority 0. When the first completion routine is entered, COMPRD, the priority would change to 1 and would never drop back to 0 during the remainder of the program. Therefore, the write will be initiated at 1, not 0. However, if it were desired to have completion routines run higher, the cp in the write request would have to be 2 (illegal in jobs) and each subsequent request would have to have a higher completion priority. This would be rather sloppy in the foreground.

This method could not be used under the utility system because interrupts would be locked out at the beginning of the first completion routine. The schedule method could not be used under utility either, because the schedule request is not available under utility.

8.3.2.15 Examples of Programs Using I/O Requests

The following two programs read one card into a buffer (BUF) from the card reader, logical unit 12. They then print the card on the teletypewriter. The first program uses system macro requests; the second program codes the system calls.

```
         JOB
0001                                NAM        CARD TO PRINT
0002                      *
0003                      *THIS PROGRAM USES SYSTEM REQUESTS TO READ AND WRITE
0004                      *
0005                                ENT        START, PRINT
0006                                EXT        IOERR ←——— error subroutine
0007  P0000  0028         BUF       BSS        BUF(40) ←——— data buffer
0008                      *
0009                      START     FREAD  12, COMPRD, BUF, 40, A, 0, 1, , 0    Formatted Read Macro
0009  P0028 |54F4  |                       ↗       ↑        ↑    ↑  ↑ ↑ ↑   ↑
0009  P0029 |0801  |              LUN 12   COMPL            |    |  | | |   X bit = 0
0009  P002A |0031 P|                       ADDR     Buffer  |    |  | | CP = 1
      P002B |0000  |                                        |    |  | RP = 0
0009  P002C |100C  |                                        |    ASCII mode
0009  P002D |0028  |                                        40 words
      P002E |0000 P|      Read is initiated
0010                      EXIT                                          EXIT Macro
0010  P002F |54F4  |      Jump to dispatcher (EXIT Request)
0010  P0030 |0A00  |      No work to do until Read completed
```

Read Completion Routine at Priority 1

```
0011  Read completed *
0012  P0031  0162   ↘COMPRD     SQP        SCHPRT        Q15 = 0 if no Read error
0013  P0032  5400  X            RTJ        IOERR         Q15 = 1 if error
      P0033  7FFF  X
0014                      SCHPRT    SCHDLE     PRINT, 0, 0                  Schedule Macro
0014  P0034 |54F4  |      Schedule PRINT at         ↑  X bit = 0
0014  P0035 |1200  |      priority 0 before exit    RP = 0
0014  P0036 |0039 P|      from completion routine
0015                                EXIT
0015  P0037 |54F4  |      Jump to dispatcher at end of completion
0015  P0038 |0A00  |      Dispatcher will now go execute PRINT at priority 0
0016                      *
0017                      PRINT     FWRITE     $FC, COMPPR, BUF, 35, A, 0, 1, I, 0    Formatted Write  Macro
0017  P0039 |54F4  |                              ↑       ↑      ↑    ↑   ↑  ↑  ↑  ↑
0017  P003A |0C01  |                       Contains               |    |   |  |  |  X bit = 0
0017  P003B |0042 P|                         LUN    COMPL         |    |   |  |  Indirect bit for LUN
      P003C |0000  |                                ADDR          |    |   |  CP = 1
0017  P003D |18FC  |                                      Buffer  |    |   RP = 0
0017  P003E |0023  |      Write is initiated                      |    ASCII mode
      P003F |0000 P|                                         35 words (on TTY)
0018                                EXIT                                     EXIT Macro
0018  P0040 |54F4  |      Jump to dispatcher
0018  P0041 |0A00  |      No work to do until Write completed
```

Write Completion

```
0019                      *
0020  P0042  0162         COMPPR    SQP        FINI          NO  IO  ERR
0021  P0043  5400  X                RTJ        IOERR
      P0044  0033  X
0022                      FINI      EXIT                                     EXIT Macro
0022  P0045 |54F4  |
0022  P0046 |0A00  |
0023                                END        START
   Entry points
 I        00FF  START    0028P  PRINT   0039P  BUF     0000P  COMPRD   0031P
 SCHPRT   0034P COMPPR   0042P  FINI    0045P  IOERR   0044X
_______________________

 0001                                NAM      IOERR      I/O subroutine
 0002                                ENT      IOERR
 0003   P0000  18FF      IOERR       0        0
 0004                                NUM      $18FF ←——— Hang instruction used for
                                     END                  checkout
_______________________

      CARD       2210  }
      IOERR      2257  }  where pgms loaded
_______________________

 I       00FF    IOERR    0000P
```

8.3.2.15

Reads and writes initiate I/O. Control returns to next instruction in program before I/O is done. Job I/O is done at priority 0. Program should not loop waiting for I/O to be done.

Completion address is entered when I/O is done, at priority 1. It should be short and exit to dispatcher. Check Q for I/O errors.

This JOB does not check Q after Request, to see if Request accepted (Q15 = 0). Jobs do not have to but system programs must.

TTY Printout

```
*P
J
*ASSEM
J
*P
J
*L,8
J
*X,, <-- No MAP
THIS PROGRAM WORKS ON THE 1700 COMPUTER SYSTEM UNDER MSOS 2.0  Formatted Write
J
```

For program checkout, hang instruction could also be used to see if completion routine entered.

```
0019                *
0020  P0042  0172   COMPPR      SQM     HANG
0021                            EXIT
0021  P0043  54F4
0021  P0044  0A00
0022  P0045  18FF   HANG        NUM     $18FF
0023                            END     START
```

EXAMPLE READ/WRITE PROGRAM

```
JOB

0001                              NAM     CARD TO PRINT
0002                              ENT     START, PRINT
0003                              EXT     IOERR
0004         00EA                 EQU     ADISP($EA)
0005  P0000  0000     START        0      0
0006  P0001  54F4                 RTJ-    ($F4)           Initiate FREAD
0007  P0002 |0201 |               NUM     $0201           READ, CP = 1  RP = 0
0008  P0003 |0009 | P             ADC     COMPRD ←——————— Completion Address
0009  P0004 |0000 |               NUM     0, $100C        THREAD, LUN  CR = 12, ASCII
      P0005 |100C |
0010  P0006 |0028 |               NUM     40              ONE CARD TO READ
0011  P0007 |001F | P             ADC     BUF             FWA BUFFER AREA
```

Control returns beneath parameter string after Read is initiated. If program has nothing to do, it should exit until completion routine is entered. Unprotected program can exit to dispatcher.

```
0012  P0008  14EA                 JMP-    (ADISP)
```

When Read is finished control will go to completion address COMPRD. Completion routine should check bit 15 of Q for I/O errors and exit to dispatcher.

```
0013  P0009  0162     COMPRD      SQP     SCHPRT
0014  P000A  5400 X               RTJ     IOERR
      P000B  7FFF X
0015  P000C  54F4     SCHPRT       RTJ-     ($F4)         Schedule PRINT at priority 0
0016  P000D |1200 |               NUM     $1200           before exit, to drop priority
0017  P000E |0011 | P             ADC     PRINT           back to 0.
0018  P000F  54F4                 RTJ-    ($F4)
0019  P0010 | 0A00 |              NUM     $A00        EXIT REQUEST
```

Dispatcher will pass control to Print after read completion routine exit.

```
0020  P0011  54F4     PRINT       RTJ-    ($F4)
0021  P0012 |0401 |               NUM     $0401           PRINT, CP = 1  RP = 0
0022  P0013 |001A | P             ADC     COMPPR    COMPLETION ADDRESS
0023  P0014 |0000 |               NUM     0, $1009, 35
      P0015 |1009 |                                       35 words on TTY
      P0016 |0023 |
0024  P0017 |001F | P             ADC     BUF             FWA BUFFER
```

Control returns here after print is initiated. Exit Request is same as jump to dispatcher.

```
0025  P0018  54F4                 RTJ-    ($F4)
0026  P0019 |0A00 |               NUM     $A00
```

After print is done, control goes to COMPPR at priority 1. Here exit from program.

```
0027  P001A  0162     COMPPR       SQP    FINI
0028  P001B  5400 X               RTJ     IOERR
      P001C  000B X
0029  P001D  54F4     FINI        RTJ-    ($F4)
0030  P001E |0A00 |               NUM     $0A00           EXIT WHEN THRU
0031  P001F  0028     BUF         BSS     BUF(40)
0032                              END     START
```

8.3.2.15

TTY Printout. Read/Write Program

Not using system macros.

Coding requests as in example will produce much faster assembly time than when using macros.

```
*P
J
*ASSEM
J
*P
J
*L,8
J
*X,N
```

The E message noted unpatched externals (IOERR). The * CR typed by operator said to ignore them.

```
THIS PROGRAM WORKS ON THE 1700 COMPUTER SYSTEM UNDER MSOS 2.0
J
```

output from WRITE

Unformatted Write did not do line feed as there are no line feed/CR in unformatted Write.

Formatted Write should be used on print devices such as TTY and LP.

Unformatted Write on printer does not work correctly. The printer buffer is filled but the line is not printed until the next output line is sent to the printer.

Read formatted ASCII data records from paper tape into BUF area and type it out. Assume the BUF area is larger than the formatted record. Continue until the reader runs out of tape, giving an error.

```
         NAM
         ENT      BEGIN
         BSS      BUF(30),STAT(1)
FLAG     ADC      0                  Flag for switching
BEGIN    0        0
START    CLR      A
         STA*     FLAG               Clear flag
READIT   RTJ-     ($F4)              Format read LU2 is paper tape
         NUM      $800               reader where to go when fin-
         ADC      READY              ished
         NUM      0,$1002,30
         ADC      BUF
         LDA*     FLAG               Hang until flag set by comple-
         SAN      1                  tion routine
         JMP*     *-2
         LDQ      STAT
         SQP      MORE-*-1
         RTJ-     ($F4)              Exit request on input error-
         NUM      $A00               end job
MORE     CLR      A                  Clear flag
         STA*     FLAG               Hang until flag set by comple-
         RTJ-     ($F4)              tion routine. Type out format
         NUM      $C00
         ADC      READY
         NUM      0,$1004,30
         ADC      BUF
         LDA*     FLAG
         SAN      1
         JMP*     *-2
         JMP*     START
READY    RAO*     FLAG               Completion routine sets flag
         STQ      STAT               Exit back to program
         RTJ-     ($F4)
         NUM      $A00
```

8.3.3 Schedule Request - Request Code 9

The schedule request is available only under MSOS.

Programs occur in the 1700 run at 16 different priorities, 15 (high) to 0 (low). A program can schedule a section of coding to be executed at a certain priority level. The address of the scheduled coding can be either in the scheduling program, external to it, or in the system.

If the desired priority of the scheduled program is higher than the running priority of the scheduling program, a pseudo interrupt occurs and the scheduled program is executed immediately. In this way a schedule request can be considered a jump that also changes the running priority. Control will return to the "interrupted" program when the scheduled program exits.

If the desired priority of the scheduled program is not higher, it is threaded onto a queue of programs waiting to be executed, in order of priority. In this way one program may schedule another to be executed at a different priority after the scheduling program exits.

Background programs can be scheduled to run at levels 0 and 1 only.

A parameter may be passed in the requestor's Q register to the program being scheduled.

8.3.3.1 Format of Schedule Request

REQ RTJ- ($F4)

15	14					9	8	7			4	3	0
0	0	0	1	0	0	1	x	0	0	0	0	p	
							c Completion Address						

rc request code, 6 bits: 9
x relative/indirect indicator, 1 bit: 0 or 1
rp this field is ignored by the scheduler
p priority, 4 bits: 0-15
c address of program scheduled

The macro call is:

```
REQ          SCHDLE       c,p,x
```

REQ will be on the 54F4, which will be generated by the macro. REQ+1 is the first word of the parameter string.

An example of a coded call would be:

```
RTJ-          ($F4)
NUM           $1201
ADC           PGM
```

(schedules PGM at priority 1)

8.3.3.2 Request Code

This field is always 9 for schedule requests.

8.3.3.3 X Bit

The x bit is used in conjunction with the c parameter. It will be discussed as it relates to this parameter.

8.3.3.4 Priority

The p field is the priority at which the programmer desires to run the scheduled program. The priority may be greater than, equal to or less than the program that schedules it. However, in the background it is always 0 or 1.

8.3.3.5 Address

The c field contains the address of the program being scheduled. The x parameter determines the forms the c may take as it did for the c parameter of the read/write requests.

x	c	The meaning of c
0 or blank	c	c is the address. i.e., SCHDLE PGM,-,- PGM is assembled as an absolute address (program relocatable).
≠0 ≠blank	c	c is relative. c is a positive increment added to the address of the first word of the parameter list to locate the address. i.e., SCHDLE PGM-*+1,-,X In this example PGM is the address.
-	(c)	c is an index to the system directory. The ( ) set bit 15. x has no meaning.

8.3.3.5

Note that there is no indirect form and that the option (c) cannot be used in the background because programs in the system library may not be scheduled from the background; see Chapter 11.

8.3.3.6 Example of Schedule Request

The following example program writes one message from program SCHEDULE and then schedules TWO (the entry point in program NEXT) to run at priority 1 and write another message.

```
0001                       NAM  SCHEDULE
0002                       ENT  XA
0003                       EXT  TWO
0004  P0000 4558    MA     ALF  *,EXAMPLE NO 1.*
      P0001 414D
      P0002 504C
      P0003 4520
      P0004 4E4F
      P0005 2031
      P0006 2E20
0005                XA     FWRITE   4,XB,MA,6,A,0,1,A
0005  P0007 54F4
0005  P0008 0C01
0005  P0009 0010  P
      P000A 0000
0005  P000B 1004
0005  P000C 0006
      P000D 0000  P
0006                       EXIT
0006  P000E 54F4
0006  P000F 0A00
0007                XB     SCHDLE   TWO,1
0007  P0010 54F4
0007  P0011 1201
0007  P0012 7FFF X
0008                       EXIT
0008  P0013 54F4
0008  P0014 0A00
0009                       END  XC
I           00FF    XA    0007P  MA      0000P  XB      0010P  TWO        0012X

0001                       NAM  NEXT
0002                       ENT  TWO
0003  P0000 4558    MB     ALF  *,EXAMPLE NO. 2*
      P0001 414D
      P0002 504C
      P0003 4520
      P0004 4E4F
      P0005 2E20
      P0006 3220
0004                TWO    FWRITE   4,XC,MB,7,A,0,1
0004  P0007 54F4
0004  P0008 0C01
0004  P0009 0010  P
      P000A 0000
0004  P000B 1004
0004  P000C 0007
      P000D 0000  P
0005                       EXIT

1 ENTRY POINT TABLE-
    XA        2E2F     TWO     2F44

J
*X
EXAMPLE NO 1.
EXAMPLE NO. 2
```

8.3.4 TIMER Request - Request Code 8

The timer request is available only under MSOS.

A hardware timing device such as the 1573 timer is required for the timer request to work since there is no real time clock in the CPU.

The timer request is a scheduled request where the program is scheduled after a predetermined time delay. The delay allowed will be from 1/60 second to 32,767 minutes.

Parameters c, p, and x are specified as for the SCHDLE request. However, instead of a parameter a time delay is specified in Q when the request is made. The delay is specified in multiples of the basic unit of the timing device. The timer passes the current contents of the core clock (E8) to the scheduled program in Q.

Timer requests are stacked in the schedule request stack but are not threaded with them. Instead, they are threaded together on the basis of time until activation. When the delay for a timer request has expired, a SCHDLE request is made by the system and the request is rethreaded into the SCHDLE thread.

The timer request is normally made by protected programs but it can be made by jobs at levels 0 or 1.

The timer was different under MSOS 1.0.

8.3.4.1 Format for the TIMER Request

RTJ- ($F4)

<table>
<tr><td>15
0</td><td>14 rc 9
8</td><td>8
x</td><td>7 u 4
0-3</td><td>3 p 0
0-15</td></tr>
<tr><td colspan="5">c</td></tr>
<tr><td colspan="5">Q</td></tr>
</table>

rc request code, 6 bits: 8
x relative/indirect indicator, 1 bit: 0 or 1
u type of units desired, 4 bits: 0 to 3
p priority, 4 bits: 0-15
c address of program to be scheduled
Q number of units desired: 1-32,767
must also be in Q register

The macro form for this request is:

```
TIMER    c,p,x,Q,u
```

An example of a coded call would be:

```
LDQ       =N$0005
RTJ-      ($F4)
NUM       $1024
ADC       PGM
NUM       5
```

(schedule PGM in 5 seconds to run at priority 4)

8.3.4.2 Request Code

Request code is 8.

8.3.4.3 X Bit

The x bit is used in conjunction with the c parameter. It will be discussed as it relates to this parameter.

8.3.4.4 Units

The type of units requested is specified based on the basic unit of the timing device.

u	
0	basic units: 60/sec on 1573
1	1/10 second
2	seconds
3	minutes

The units field is used in conjunction with the Q parameter to calculate the desired time delay and it controls the precision required in the timing.

8.3.4.5 Priority

This is the priority at which the scheduled program is to run. It follows the same rules as in a schedule request.

8.3.4.6 Address

The c field contains the address of the program being scheduled and it follows the same rules as for a schedule request.

8.3.4.7 Q parameter

The Q parameter is the number of the type of units desired; it must be in the Q register and the Q word of the request.

8.3.4.7

For example, a timer request for 5 seconds could be either u=2 and Q=5, or u=1 and Q=50. If u=2, the request will be on the seconds thread and will be scheduled in at least 5, but less than 6, seconds. If u=1, the request will be on the 1/10 second thread and will be scheduled in at least 5 seconds, but less than 5 1/10 seconds. This is how the desired timing precision can be achieved.

8.3.4.8 Example of TIMER Request

Assume the basic unit of the system is 1/60 second. Schedule the program PGM using an absolute call, after 20 minutes has elapsed, to run at priority 6.

```
        ENQ             20
        TIMER           PGM,6,,20,3
```

8.3.5 STATUS Request - Request Code 3

This request is available to unprotected programs only. Foreground programs do not use the status request to obtain status.

The status of a particular read/write request is obtained with the status request. The status of the request is returned in the A, Q, and I registers.

The status request can be used to determine whether an I/O operation is complete, to examine the type of hardware to which the logical unit is assigned, to check the dynamic status on the hardware, or to find out how far along the I/O is by checking the current buffer address.

8.3.5.1 Format of the STATUS Request

RTJ- ($F4)

15	14		rc			9	8	7							0
0	0	0	0	0	1	1	x	0	0	0	0	0	0	0	0
15			12	11	10	9									0
0	0	0	0	A		LUN									
ap address of parameter list															

rc request code, 6 bits: 3

x relative/indirect indicator, 1 bit: 0 or 1

ℓ logical unit, 10 bits; modified by a, same as for read/write requests

a logical unit address indicator, 2 bits; same as for read/write requests
Settings are:

a = blank = 00: ℓ is the logical unit number
a = R = 01: ℓ is a signed increment (±1FF)
a = I = 10: ℓ is a core address ($0<\ell\leq 3FF$)

ap address of parameter list of request

The macro call is:

```
          STATUS       ℓ,ap,a,x
```

An example of a coded call would be:

```
          RTJ-         ($F4)
          NUM          $0600
          NUM          0004
          ADC          REQ+1
```

(Obtain status of a request on logical unit 4; request address is REQ+1)

8.3.5.2 Request Code

The request code is 3.

8.3.5.3 X Bit

The x bit is used in conjunction with the ap parameter and will be discussed as it relates to that parameter.

8.3.5.4 A Field

The a bits modify the logical unit field in the same way as in the I/O request.

8.3.5.5 Logical Unit

The logical unit is the same logical unit upon which the I/O request was made (the one we wish to have the status of). The logical unit may be specified directly, indirectly or relatively, in the same way the logical unit was specified in the I/O request.

8.3.5.6 Address of Parameter List

The ap field is the address of the parameter list for the request for which status is desired. It can be specified directly, indirectly or relatively, in conjunction with the x bit.

x	ap	Meaning of ap
=0 or blank	ap	ap is the address of the first word of the parameter list of the I/O request. i.e., STATUS -,REQ+1,-,- REQ+1 is the parameter address.
≠0 ≠blank	ap	ap is relative. ap is a positive increment added to the address of the first word of the status request parameter list to obtain the address of the first word of an input/output request parameter list.

x	ap	Meaning of ap
		i.e., STATUS -,REQ+1-*+2,-,X REQ+1 is the parameter address.
=0 =blank	(ap)	ap is indirect ap is the address of a location containing the address of the first word of an input/output parameter list. i.e., STATUS -,(REQADR),-,- REQADR contains the address of REQ+1
≠0 ≠blank	(ap)	ap is double relative. ap is a positive increment added to the address of the first word of the status request parameter list to obtain the address of a location containing another positive increment. The second increment is added to the address of the first word of the status request parameter list to obtain the address of the first word of an input/output request parameter list. Because of wraparound in adding, both increments may refer to locations ahead of or behind the status request. i.e., STATUS -,(REQREL-*+2),-,X REQREL contains a number to be added to the first word of the status parameter list to form the distance to REQ+1 (s+1+contents of REQREL).

Although the parameter address is not needed in the utility system, it is supplied to be compatible with 1700 MSOS. If it is omitted, only device status will be returned.

8.3.5.7 Reply to STATUS Request

Following execution of the status request, the A, Q, and I registers contain the status. The content of these registers is as follows:

A: hardware status of device
Q: word 8 of Physical Device Table for device
I: last core address of data transmission

8.3.5.7.1 Hardware Status

A		Hardware Status Reply

The hardware reply is dynamic unless the device is connected to a buffered data channel and the channel is busy. If it is not busy, the hardware reply is the status obtained at the completion of the last request for that device. For an explanation of hardware replies refer to the hardware specifications.

This status is the hardware reply bits on the device itself.

8.3.5.7.2 Word 8 of PDT

See appendix D of the MSOS Reference Manual for contents of this word.

	15 14	13 11	10 4	3 1	0
Q	E	S	T	R	P

Word 8 EREQST

Request Status

	Bits	
P	0 = 1	Device not available to unprotected programs
R	1 = 1	Device may be read from unprotected programs
	2 = 1	Device may be written by unprotected programs
	3 = 1	Equipment table includes words 18-33 for message buffering
T	4-10	Equipment type constant (T), see T table
S	11 12 13	Equipment Class, see S table
E	14 = 1	Device failure
	15 = 1	Operation is in progress
	= 0	Operation is complete

S TABLE

EQUIPMENT CLASS CODES

Word 8, EREQST Bits 11-13

0	Class not defined
1	Magnetic tape device
2	Mass storage device
3	Card device
4	Paper tape device
5	Printer device
6	Teletype device
7	Reserved for future use

8.3.5.7.2

T TABLE

STANDARD EQUIPMENT TYPE CODES

Word 8, EREQST Bits 4-10

0	1711/1713 Teletypewriter
1	1721/1722 Paper Tape Reader
2	1723/1724 Paper Tape Punch
3	Unassigned
4	
5	1738/853 Disk Unit
6	1751 Drum Unit
7	1729 Card Reader
8	1738/854 Disk Unit
9	601 Magnetic Tape Unit
10	Software Buffering Device
11	1742 Line Printer
12	1728/430 Card Reader/Punch
13	Software Core Allocator
14	210 CRT Display Station
15	1558 Latching Relay Output
16	1553 External Register Output
17	311B/312B Data Set Terminal
18	322/323 Teletype Terminal
19	Unassigned
20	166 Line Printer
21	1612 Line Printer
22	415 Card Punch
23	405 Card Reader
24	608 Magnetic Tape Unit
25	609 Magnetic Tape Unit
26	1713 Teletype Keyboard
27	1713 TTY Paper Tape Punch
28	1713 TTY Paper Tape Reader
29	Unassigned
30	1797 Buffered I/O Interface
31	Software Dummy Alternate
32	1584 Selectric I/O Typer
33	1582 Flexowriter I/O Typer
34	1716 Coupling Data Channel
35	1718 Satellite Coupler
36	Unassigned
37	8000 Series Magnetic Tape Unit
38	Unassigned
39	
40	1530 A/D Converter 30/40 PPS
41	1534 A/D Converter 200 PPS
42	1538 A/D Converter High Speed
43	Unassigned
44	
45-99	Reserved for future standard equipment
100-127	Open for user assignment

The word 8 status can be used to determine several things about the device. The E field can be used to see if any request is active on the device and whether a hardware error is present; this requires operator intervention. T can be used to find out what kind of equipment the device is. For example, a program could see if the standard output device is a teletypewriter or a line printer and then output either 70-character lines or 136-character lines. The P and R fields can be used to determine the availability to jobs.

8.3.5.7.3 Current Buffer Address

I	address

The address of the last word that was stored in the buffer or written from the buffer is in the I register. In this way a job can determine how much of its buffer has been filled during operation.

8.3.5.8 Example of Status Request

```
0015                   AA      FWRITE   $FC,XA,(ADDR),(LENGTH),A,0,1,I
0015   P008F  54F4
0015   P0090  0C01
0015   P0091  00A1  P
       P0092  0000
0015   P0093  18FC
0015   P0094  8053  P
       P0095  8052  P
                       XA      EXIT
                               SQP      OK-*-1
0022                   OUT
0022   P009C  54F4             STATUS   $FC,AA+1,I
0022   P009D  0600
0022   P009E  80FC

                       OK      ⋮
                               EXIT
```

The above example takes status on the request from the completion routine if an error indication is present.

More examples of status requests are in the CKASSM routine, section 8.3.10.

8.3.6 GTFILE Request - Request Code 13

This request is available only to background programs and only under MSOS.

A permanent file that has been placed in the program library* can be accessed during execution by the GTFILE request. A file is brought into core as it appears on the mass storage device; GTFILE does not load a program. If a program is

*A file is placed in the program library by a LIBEDT operation.

placed in the program library as a file, it must be in its absolute binary form. Data files cannot be changed and written back on the library during execution. GTFILE only reads the file into core.

8.3.6.1 Format of GTFILE Request

<table>
<tr><td colspan="8">RTJ-</td><td colspan="2">($F4)</td></tr>
<tr><td>15
0</td><td colspan="6">14 rc 9
0 0 1 1 0 1</td><td>8
x</td><td>7 4
rp</td><td>3 0
cp</td></tr>
<tr><td colspan="10">c
completion address</td></tr>
<tr><td colspan="10">thread</td></tr>
<tr><td colspan="3">v</td><td colspan="2">0
m</td><td colspan="2">2
a</td><td colspan="3">$C2
ℓ</td></tr>
<tr><td colspan="10">w1</td></tr>
<tr><td colspan="10">s
starting core address</td></tr>
<tr><td colspan="10">w2</td></tr>
<tr><td colspan="10">f
filename address</td></tr>
</table>

rc	request code, 6 bits: 13
x	relative indicator
rp	request priority (for MSOS), 4 bits: 0
cp	completion priority, 4 bits: 1
c	completion address
thread	for system: 0
v	error bits, 3 bits
m	mode, 1 bit: 0 *
a	logical unit modifier, 2 bits: 2 *
ℓ	logical unit address of MS device, 9 bits: $C2 *
w1	first word desired in file
s	starting core address for buffer
w2	last word desired in file
f	filename address

The macro call is:

```
GTFILE  c,f,s,w1,w2,x,rp,cp
```

*Set by assembler if macro call used.

An example of a coded call would be:

```
FILNAM      ALF        *,FILE  *       FILE NAME
             {
            RTJ-       ($F4)
            NUM        $1A01
            ADC        GOT             COMPL
            NUM        0
            NUM        $08C2           BINARY,LIBUNIT
            NUM        0               WHOLE FILE WANTED
            ADC        BUF             BUFFER
            NUM        0
            ADC        (FILNAM)        ADDR OF FILE NAME
```

(The file FILE is requested to be stored in BUF. GOT is the completion address.)

8.3.6.2 Request Code

Request code is 13.

8.3.6.3 X Bit

The x bit is used in conjunction with the f parameter. It will be discussed as it relates to that parameter.

8.3.6.4 Request Priority

The request priority is always 0 in the background for jobs. It will be used as the request priority for the mass storage driver when it reads in the file.

8.3.6.5 Completion Priority

The completion priority is always 1 in the background.

8.3.6.6 Completion Address

c is for the completion address and takes the same form as for the read/write request.

8.3.6.7 Mode

The mode field must be set to 0 (binary).

8.3.6.8 A Field

The a bit, logical unit modifier, must be set to 2 (indirect).

8.3.6.9 Logical Unit

The address of the library logical unit, $C2, must be set in this field.

8.3.6.10 Word Addresses: w1, w2

w1 and w2 are the beginning and ending word addresses (mass storage) within the file if word addressing is used and the disk word driver is present in the system.

If only a portion of a file is wanted, the w1, w2 specifies the words wanted. They are specified directly. If the complete file is wanted, w1 and w2 should be left blank.

```
GTFILE  -,-,-,10,45,-,-

GTFILE  -,-,-,,,-,-
```

In the first example words 10 to 45 would be brought in. In the second, the complete file would be brought in.

8.3.6.11 Starting Core Address

s is the starting address of the block into which the file or a portion of the file is to be transferred. x determines the type of addressing mode s takes.

x	s	Meaning of s
-	s	s is the starting address; x has no meaning. e.g., GTFILE -,-,BUF,-,-,-,-,-
=0 =blank	(s)	s is indirect. s is the core location which contains the starting address of the block. e.g., GTFILE -,-,(BUFADR),-,-,-,-,- In this case BUFADR contains the address of the block.
≠0 ≠blank	(s)	s is relative. s is a positive increment added to the first word of the parameter list to form the starting address of the block. e.g., GTFILE -,-,(BUF-*+5),-,-,X,-,- BUF is the buffer.

8.3.6.12 File Name Address

The f parameter indicates the address of the first word of a three-word block that contains the ASCII name of the file. It takes two forms, f and (f).

f is a positive increment to be added to the first word of the parameter list when it stands alone.

For example:

```
GTFILE   -,NAME-*+7,-,-,-,-,-,-
```

In this case NAME would contain the first two characters of the ASCII name and the relative distance to NAME would be assembled into the macro.

(f) indicates f is the address of the three word block containing the ASCII name.

For example:

```
GTFILE   -,(NAME),-,-,-,-,-,-
```

In this case the address of NAME is assembled into the macro.

The system searches the program library for the file with the specified name.

It is supposed to be necessary to specify two additional words at the end of the request in which the system will return the actual sector address of the file. This does not work, however, so we omit it.

8.3.6.13 Example of a GTFILE Request

The following example uses a GTFILE request to obtain a file named SYSINI from the program library and store it into a buffer, beginning at $6000 (absolute address).

It happens that in this example the GTFILE request is to obtain a file that is an absolute program and is to transfer control to it; but the GTFILE could simply have been used to input data to a buffer.

The example shows how the GTFILE works and how it is assembled.

8.3.6.13

EXAMPLE USING GTFILE REQUEST FOR SYSTEM INITIALIZER

```
0001                          NAM      GETSI
0002                          ENT      SI
0003  P0000 5359    FILNAM    ALF      *,SYSINI* ◄────── Name of FILE in program library
      P0001 5349
      P0002 4E49
0004  P0003 5349    BUF       ALF      *,SI   IN* ◄───── MSG Buffer for TTY
      P0004 2049
      P0005 4E20
0005        6000    SIADDR    EQU      SIADDR($6000)     File to go at $6000
0006  P0006 0000    SI        0        0
0007                GETFIL    GTFILE  GOT,(FILNAM),SIADDR,,,0,0,1
0007  P0007 54F4                                                    CP = 1
0007  P0008 1A01       Completion address                          RP = 0
0007  P0009 0014 P   after SYSINI is brought    Address           X bit = 0 (not blank)
      P000A 0000                 into core      where      Core
0007  P000B 08C2                                file name  address   Disk  address
      P000C 0000                                is         where     left blank;
      P000D 6000                                           file is   program li-
0007  P000E 0000                                           to go     brary will be
      P000F 8000 P                                                   searched
0008                          EXIT (wait for completion)
0008  P0010 54F4
0008  P0011 0A00
0009                          EXIT         (unnecessary)
0009  P0012 54F4
0009  P0013 0A00
0010  P0014 017B    GOT       SQM      NOGOOD
0011                SIIN      FWRITE   $FC,WROTE,BUF,3,A,0,1,I,0
0011  P0015 54F4                                                 X bit = 0 (not blank)
0011  P0016 0C01                  LUN                            Indirect bit referring to $FC
0011  P0017 001E P       (std comment    COMPL                CP = 1
      P0018 0000             device)     ADDR              RP = 0
0011  P0019 18FC                              MSG
0011  P001A 0003                              BUF       ASCII
      P001B 0003 P                                  3 words
0012                          EXIT
0012  P001C 54F4              Wait until Write is done
0012  P001D 0A00
0013  P001E 0171    WROTE     SQM      NOGOOD
0014  P001F 1CED              JMP*     (GETFIL+6) ◄───── Jump to beginning address of
0015                NOGOOD    EXIT                       SYSINI which is $6000
0015  P0020 54F4
0015  P0021 0A00
0016                          END      SI
```

This program may be reassembled for any system. Change the EQU for the desired high core address where the system initializer is to be placed. The system initializer should be stored in the program library under the file name SYSINI. It can then be called into core by typing on the TTY

*SI

SYSINI was made a file so it could be stored in high core.

8.3.7 LOADER Request - Request Code 7

This request may be made only by background programs.

The loader request enables the program to load programs during execution. A program is loaded beginning at the first word of unassigned, unprotected core. When loading, the loader resides in the upper part of unprotected core, wiping out COMMON if it was being used.

The parameters for the loader request are in the A and Q registers. These parameters prescribe what type of load is to take place and from which logical unit.

	15	3 0
A	ℓu	t

	15 0
Q	tna

t type of loading operation; discussion follows

ℓu logical unit number of the input unit if a relocatable binary program is being loaded

tna entry point, core address of the first of three sequential locations containing the entry point name

t	Function	ℓu	tna
0	Load relocatable binary programs from any unit	input device	ignored
1	Load from program library on library unit	library unit	ignored
2	Load program from library unit and execute immediately	library unit	location of program name
3	Produce memory map	ignored	ignored
4	Look up entry point name	ignored	location of entry point name
5	Same as t = 1 but no memory map printed		
6	Search directory of core-resident entry points	ignored	ignored
7	Initialize data base	ignored	ignored

When the load is completed without an error, the A register contains the last transfer address given, as in normal loading. If an error terminated loading, A contains zero and the Q register contains the storage address of the input block processed by the loader at the time the error occurred.

8.3.7.1 Format of the LOADER Request

			RTJ-				($F4)	
15	14					9	8	0
0	0	0	0	1	1	1	0 ———	0

The macro call would be:

```
LOADER
```

An example of a coded request would be:

```
RTJ-        ($F4)
NUM         $0E00
```

See the core request for a program example using the loader request.

8.3.8 CORE Request - Request Code 11

The core request can be made only by background programs.

The core request can expand or contract available unprotected core. For example, if during execution the high locations of one's program are no longer needed, one could release these locations by a core request so that another program could be loaded into this area.

The core request has two forms, depending on the contents of the A and Q registers. If A and Q are zero, the core request asks for the current boundaries of unassigned unprotected core. When the request has been processed, the Q register contains the lower boundary-1 and the A register contains the upper boundary+1. (The contents of A and Q are actually obtained from core locations $ED and $EC.) With this information the program can set new boundaries to available unprotected core.

When A and Q are non-zero and a core request is made, the new boundaries are set according to the contents of A and Q. A contains the new upper boundary and Q the lower. Both boundaries must be within unprotected core and A must be larger than Q.

The core request is supposed to return the actual lower and upper bounds of unprotected core, but since it currently returns the lower-1 and upper+1 (from $ED and $EC) we program it to allow for that.

8.3.8.1 Format of Core Request

RTJ- ($F4)

15	14		rc			9	8								0
0	0	0	1	0	1	1	0	0	0	0	0	0	0	0	0

The macro call would be:

CORE

The coded call would be:

```
RTJ-        ($F4)
NUM         $1600
```

8.3.8.2 Example of Core Request

The following example program, LOADERC, makes a core request to obtain bounds. It then drops the lower bound by 22_{16}. Then it makes a loader request to load a relocatable binary program tape from logical unit 2. The program on the tape will begin to overlay the BSS block in LOADERC at address P0011. LOADERC receives the entry point address in A from the loader (from the end tna card of the loaded program) and stores it in the second word of the jump instruction, where it can jump to the loaded program.

```
                          NAM   LOADERC
                          EXT   ENTRY          Unpatched external
       0002               EQU   TAPEU(2)
P0000  0846      XX       CLR   A,Q
                          CORE                 Get core limits
P0001  54F4
P0002  1600
P0003  0DDD                 INQ    -$22
                          CORE                 Set lower limit
P0004  54F4
P0005  1600
P0006  C000               LDA   =XTAPEU        Set up LUN
P0007  0002
P0008  0FC4               ALS   4
                          LOADER               Call loader
P0009  54F4
P000A  0E00
P000B  6802                 STA*   XA+1        Save entry point address
P000C  1400      XA       JMP+  **             Jump to it
P000D  0000 P
P000E  0B00               NOP
                          EXIT
P000F  54F4
P0010  0A00
                          ENT   XX
P0011  0022                     BSS   PT($22)
                          END     XX
```

The system recovery package was used to dump some of the program area beginning at P000D in LOADERC. It shows that the new buffer of the loaded program exactly overlayed the BSS in LOADERC; then the write request, beginning at P0009 in the new program, wrote out the confirming message.

Use of the system to execute this job is covered in Chapter 9, but the example is included here so that it can be studied for later reference (since it applies to the core and loader requests).

RE DUMP

```
                                                          Buffer PT began here at 2F3F but is now overlayed
                                                          by message buffer.  CORE wiped out PT buffer and
                                                          set new lower limit at 2F3F.
                     New program F was
                     loaded at 2F3F.                                   ANOTHER EXAMPLE                        Write request at
                                                                                                              entry point of F
          F          2F3F
         (2F3B       2F48  0B00  54F4  0A00  414E  4F54  4845  5220  4558  414D  504C  4520  0008  54F4  0C01  2F51
 RE DUMP (2F4B       0000  1004  AF47  2F3F  54F4  0A00  0179  54F4  0D01  0009  0000  1004  FFF3  7FEB  54F4  0A00
         (2F5B       1804
                                            Buffer address in F
```

TTY PRINTOUT

```
*P
J
*L, 8
J
*SR
J
*X                                  Unpatched external ENTRY which was not needed so
                                    ignored.
E *
L, 02 FAILED 02                     Load F from paper tape reader.  There is no *T on
ACTION                              the end of the tape, hence the message and CU.
CU
ANOTHER EXAMPLE                     Output from Program F
ANOTHER EXAMPLE

RE
*2F3B
ERR

RE
*D2F3B, 2F5B     <-------------     RE Dump Request

RE
```

8.3.9 INDIR Request - No Request Code

This request can be used by foreground or background programs under MSOS only. It is not a separate request but is an indirect version of any other request.

Any request can be used again without repeating the request by using the INDIR request.

8.3.9.1 Format of the INDIR Request

```
              RTJ-          ($F4)
```

15	0
1	ap

Only in the INDIR request should bit 15 of the first word of the parameter list be set to 1. This tells the system that the word under the RTJ- ($F4) is not a parameter but is the address of the parameter list to be requested.

The macro form is:

```
              INDIR    (p)
```

p is the address of the parameter list; it must be in parentheses.

To code the call:

```
              RTJ-          ($F4)
              ADC           (REQ+1)
```

(if the desired request parameters begin at REQ+1)

8.3.9.2 Example of the INDIR Request

An example of the INDIR request would be one in which the request parameters could be stored in a buffer and an indirect request could cause them to be executed.

The following example stores the buffer address for MESSAGE in the s field of a request at REQBUF (the number of words in n) and then executes the request at REQBUF.

Note that by using the INDIR request, control returns beneath it after the request is initiated. If a jump had been made to REQBUF-1 (if a RTJ- ($F4) were there), control would return under the parameter string at REQBUF. This may not be the desired action.

```
                  ≀
BUF               ALF          *,MESSAGE  *
                  ≀
                  LDA          =XBUF
                  STA*         S
                  ENA          4
                  STA*         N
                  INDIR        (REQBUF)
                  ≀
REQBUF            NUM          $0C01       FWRITE
                  ADC          COMPL
                  NUM          0
                  NUM          $1004       ASCII,TTY
N                 NUM          0
S                 NUM          0
                  ≀
```

More examples of the INDIR request appear in the routine CKASSM in the next section, 8.4.

8.4 PROBLEM

The following program is a routine which can be used to check out the macro assembler. Study it carefully to see what it does.

After studying the program as it is written, figure out what would happen if the two SQP instructions at P0007 and P000D were SQN instead.

Comprehension of the CKASSM routine should be considered a "final examination" on requests. Any points which are not thoroughly clear to the reader should be restudied carefully in the appropriate sections. A very good knowledge of these requests is required before the student goes on to study Part II of the training manual.

```
0001                     NAM  CKASSM          84758702
0002              *      DECK ID   84758702   JUNE 4, 1968
0003              *      VERIFICATION TEST FOR A007 MACRO ASSEMBLER
0004              *      SPEC ID REFER TO 84800300
0005              *      PROGRAMMING SYSTEMS, A/D SYSTEMS DIVISION, CDC
0006              *      ASSEMBLE USING 1700 MACRO ASSEMBLER
0007              *      THIS PROGRAM IS A VERIFICATION FOR THE
0008              *      MACRO ASSEMBLER
0009              *      THIS IS ONE OF A SERIES OF TESTS THAT USES
0010              *      INDIRECT SYSTEM REQUESTS ON VARIOUS
0011              *      COMBINATIONS OF (F)READ/(F)WRITE REQUESTS.
0012                     ENT  CKASSM
0013 P0000 0B00   CKASSM NOP
0014                     INDIR (F1+1)
0014 P0001 54F4
0014 P0002 80C5 P
0015                     INDIR (W1+1)
0015 P0003 54F4
0015 P0004 80BC P
0016              IN1    INDIR (T1+1)
0016 P0005 54F4
0016 P0006 80B8 P
0017 P0007 0161          SQP  1
0018 P0008 18FC          JMP* IN1
0019                     INDIR (R1+1)
0019 P0009 54F4
0019 P000A 80AF P
0020              IN2    INDIR (T2+1)
0020 P000B 54F4
0020 P000C 80AB P
0021 P000D 0161          SQP  1
0022 P000E 18FC          JMP* IN2
0023                     FWRITE $FC,,BUF,25,B,,,I
0023 P000F 54F4
0023 P0010 0C00
0023 P0011 0000
     P0012 0000
0023 P0013 08FC
0023 P0014 0019
     P0015 004A P
0024                     EXIT
0024 P0016 54F4
0024 P0017 0A00
0025 P0018 4E45   MSG    ALF  25,NEXT MESSAGE SHOULD INDICATE VERIFICATION
     P0019 5854
     P001A 204D
     P001B 4553
     P001C 5341
     P001D 4745
     P001E 2053
     P001F 484F
     P0020 554C
     P0021 4420                             P002C 4E20
     P0022 494E                             P002D 2020
     P0023 4449                             P002E 2020
     P0024 4341                             P002F 2020
     P0025 5445                             P0030 2020
     P0026 2056
     P0027 4552
     P0028 4946
     P0029 4943
     P002A 4154
     P002B 494F
```

```
0026 P0031 4D41    MSG1    ALF   25,MACRO ASSEMBLER ON 1700 OK
     P0032 4352
     P0033 4F20
     P0034 4153
     P0035 5345
     P0036 4D42
     P0037 4C45
     P0038 5220
     P0039 4F4E
     P003A 2031
     P003B 3730
     P003C 3020
     P003D 4F4B
     P003E 2020
     P003F 2020
     P0040 2020
     P0041 2020
     P0042 2020
     P0043 2020
     P0044 2020
     P0045 2020
     P0046 2020
     P0047 2020
     P0048 2020
     P0049 2020
0027 P004A 0060            BZS   BUF(96)
0028               T2      STATUS 5,R1+1
0028 P00AA 54F4
0028 P00AB 0600
0028 P00AC 0005
0028 P00AD 00AF P
0029               R1     FREAD $C2,,BUF,25,B,,,I
0029 P00AE 54F4
0029 P00AF 0800
0029 P00B0 0000
     P00B1 0000
0029 P00B2 08C2
0029 P00B3 0019
     P00B4 004A P
0030 P00B5 0000            NUM   0,1
     P00B6 0001
0031               T1      STATUS 5,W1+1
0031 P00B7 54F4
0031 P00B8 0600
0031 P00B9 0005
0031 P00BA 00BC P
0032               W1      FWRITE $C2,,MSG1,25,B,,,I
0032 P00BB 54F4
0032 P00BC 0C00
0032 P00BD 0000
     P00BE 0000
0032 P00BF 08C2
0032 P00C0 0019
     P00C1 0031 P
0033 P00C2 0000            NUM   0,1
     P00C3 0001
```

```
0034                    F1      FWRITE $FC,,MSG,25,A,,,I
0034 P00C4 54F4
0034 P00C5 0C00
0034 P00C6 0000
     P00C7 0000
0034 P00C8 18FC
0034 P00C9 0019
     P00CA 0018 P
0035                            END   CKASSM
```

```
J
*P
J
*ASSEM
OPTIONS  LX
J
*P
J
*L, 5
J
*X, N
NEXT MESSAGE SHOULD INDICATE VERIFICATION
MACRO ASSEMBLER ON 1700 OK
J
```

CHAPTER IX

MSOS USE

9

CHAPTER IX - MSOS Use

9.1 JOB PROCESSOR

The job processor is that part of the operating system which monitors the background. It is a system program, and it allows jobs to run in the background when the system does not need the CPU or the background core area. Under the control of the job processor are the program library and such jobs as assembling, loading, compiling and executing.

The job processor resides in the system library on the mass storage device until it is called into execution by a manual interrupt and an * followed by any control statement and a carriage return (CR).

For example: The operator depresses the MI key and types *P.

```
MI
*P
J
```

The system will type the MI and J. (In this chapter all messages typed by the operator will be circled.) MI indicates that the manual interrupt has been accepted.

The J will be printed when the job processor has come into core and is ready for a control statement. The job processor types its messages on the standard comments device which, in most cases, is the teletypewriter.

Each control statement to the job processor must begin with an asterisk and must be terminated with a carriage return. The job processor will type a J when it has finished doing what it was instructed to do and is ready to accept another control statement. It will also light the BREAK light on the teletype, and the operator must depress the BREAK RELEASE key to turn off the light (and start the motor), then type his statement. If this light was not lighted, the system is not waiting for his input.

If the operator realizes that he has typed the statement incorrectly before he types the carriage return, he can "erase" it by typing a rub out, line feed, carriage return and then proceed.

JOX, hhhh indicates an error in a control statement to the job processor or a processing error in the background program. A summary of the error messages appears in Appendix D.

The following control statements are available for the operator to use to instruct the job processor in running a job. A brief description of their meaning is here and they are described in detail in the MSOS reference manual.

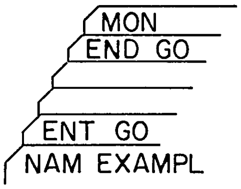
Figure 17. Control Statements Available Under the Job Processor

Control Statement	Meaning
*P	Brings in the loader, initializes for an independent loading operation.
*K, Iu, Pu, Lu	Alters standard logical units. Where I is input, P is punch and L is the list device.
*L, u	Loads a program from logical unit u.
*X, m	Executes the program that was loaded. If m is blank, the memory map will be printed.
*<entry point>	Loads a program from the program library and transfers control to it.
*Z	If a job is in process, terminates the job. Following a J, terminates job processor.
*V	Switches to standard input unit for subsequent control statements
*U	Returns control to comments unit for subsequent statements. (i.e., This statement could appear on the card reader.)
*B	Brings in the breakpoint package.
*SR	Brings in the system recovery package.
*R, u	Restores a logical unit u after it has failed.
*T	Terminates loading from input device. Should be last record of object deck being loaded.
*	Continue execution.

9.1.1 Assembling a Program

Once an Assembly Language program has been written and placed in machine readable form (punched on a card, typed in USASI font for the OCR or punched on paper tape), it may be assembled by bringing in the assembler (*ASSEM(CR)) under the job processor's control. The *ASSEM(CR) statement assumes that the source deck will be read from the standard input device. For example, if the standard input device is the card reader, the programmer could place his cards in the hopper, depress the clear button and, at the teletypewriter, bring in the job processor and assembler in the manner described on the following page.

Input Cards



Comments Device

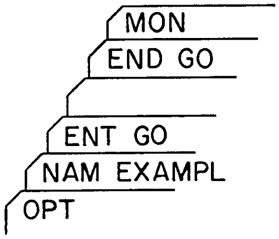
MI
*P
J
*ASSEM
J

The *P brings in the job processor and the loader to load the assembler. Assembly will begin immediately and will continue until a MON card or an illegal assembly card is encountered.

In this case an OPT card was not used; therefore, a listing will be made, an object program will be prepared on the standard output device and the relocatable binary program will be stored on the first scratch sector of the mass storage device for immediate loading and execution. This binary image of the object program on mass storage is called "load-and-go" (LGO).

If an OPT card is used, the operator will have a choice of three results: listing, punching an object program and "load-and-go". The computer will type OPTIONS.

Input



Comments Device

J
*P
J
*ASSEM
OPTIONS

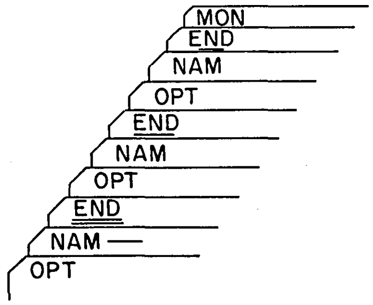
The operator's response can be: L if he wants a listing; P if he requires an object program; and X if he wants the "load-and-go" eXecute option. In the situation where a programmer is debugging and wants a listing plus the "load-and-go" option to allow an immediate load from the mass storage device, he would type the following:

*P
J
*ASSEM
OPTIONS XL
J

The X, L and P can be in any combination and in any order; e.g., LX or just P.

If more than one program is assembled in the same run and each has an OPT card, the assembler will ask for new options when it assembles each subsequent program.

Input



Comments

*P
J
*ASSEM
OPTIONS XL
OPTIONS XLP
OPTIONS XP
J

Assembling a Program, Illustration 1

```
Col.  Col.   Col.     Source
 1     2      3       Program
0001                            NAM    EXAMPL
0002                            ENT    GO
0003  P0000  4558     MESSAG    ALF    *,EXAMPLE PRINT OUT*
      P0001  414D
      P0002  504C
      P0003  4520
      P0004  5052
      P0005  494E
      P0006  5420
      P0007  4F55
      P0008  5420
0004                  GO        FWRITE   $FC,GO1,MESSAGE,9,A,0,1,I,0
0004  P0009  54F4
0004  P000A  0C01
0004  P000B  0012 P
      P000C  0000
0004  P000D  18FC
0004  P000E  0009
      P000F  0000 P
0005                            EXIT
0005  P0010  54F4
0005  P0011  0A00
0006  P0012  0806     GO1       SET    Q,A
0007                            END    GO

 I           00FF     GO        0009P   MESSAG   0000P  GO1     0012P
```

Column one is the card number, e.g., the symbol GO is on card four (4). Column two, when preceded by a P, is the address of each storage word relative to the beginning of the program. For example: the EXIT request takes up words 10_{16} and 11_{16}.

The complete program occupies 13_{16} locations (0000-0012). Note that the ENT card doesn't occupy a location in the program nor does the END card.

Column 3 shows the assembled code. For example, the first word of MESSAG assembles as 4558_{16} which is the ASCII code for the letters EX. SET Q,A in machine language is 0806_{16}. If a letter follows the machine code, for example P, the contents of the word is to be modified at load time. In the previous example, the word will be modified by the program counter. e.g., If the program is loaded at location 2000_{16} then the contents, after loading, of word $000B_{16}$ would be 2000_{16} + 0012_{16} or 2012_{16}.

Everything to the right of Column 3 is a printout of the source deck as it appeared on input.

Assembling a Program, Illustration 2

```
Col.    Col.  Col.    Source
 1       2     3      Deck
----    ----- ----    ------------------------------------------
0001                        NAM    PSEUDO   EX
0002          0010          EQU    CAT(16)
0003          0000 D        DAT    BUF(50),EX(50)
              0032 D
0004          0000 C        COM    TABLE(50)
0005    P0000 000A          BSS    XX(10),X,Y,Z
        P000A 0001
        P000B 0001
        P000C 0001
0006          0032 D        ORG    EX
0007    D0032 C400          LDA    TABLE
        D0033 0000 C
0008    D0034 F0F0          NUM    $F0F0,$FFFF
        D0035 FFFF
0009    D0036 0000 D        ADC    BUF
0010          0000 P        ORG    XX
0011    P0000 5341          ALF    *,SAM IS OK*
        P0001 4D20
        P0002 4953
        P0003 204F
        P0004 4B20
0012          000D P        ORG*
0013    P000D 0B00          NOP
0014    P000E 0B00          NOP
0015    P000F 0000          VFD    X4/CAT
0016    P0010 0B00     QQ   NOP
0017    P0011 0B00          NOP
0018                        EXIT
0018    P0012 54F4
0018    P0013 0A00
0019                        ENT    QQ
0020                        END    QQ
```

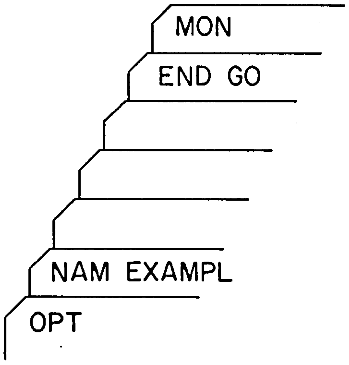
In this example a D appears in column two - specifying the DATA area of storage. Thus, location 0032 of the DATA area will contain $C400_{16}$ when loaded.

The D and C following the machine instructions in column three indicate that these instructions will be modified by either the DATA counter or the COMMON counter. An X appearing in this position indicates an address eXternal to the program. The COMMON and DATA counters are used by the loader when it is loading into these core areas.

9.1.2 Loading and Executing a Program

The general form for the load statement is *L,u(CR). The u is for the logical unit of device from which the object deck is to be loaded. The program is loaded immediately but is not placed into execution until the *X(CR) statement is given. If a "load-and-go" option was chosen at assembly time, we could load and execute the program in the following manner (given that the scratch unit is logical unit 8):

Input



```
Comments          Notes
Device
J
*P
J
*ASSEM
OPTIONS XLP
J
*P
J
*L,8
J
*X                Execute the Program
EXAMPLE PRINT     Output from the program
J
```

The following is printed on the standard list device as a result of the *L,8(CR) (program name and address where loaded):

```
List Device

EXAMPL      24E0
```

and as a result of the *X

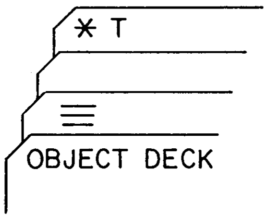
```
         List Device

ENTRY POINT TABLE  (MAP)

         GO     24E9
```

Note that it was necessary to call in the loader with a *P control statement before asking the loader to load with the *L statement. After the loader loaded the program, the printout EXAMPL 24E0 indicated that the absolute beginning core address where the program was loaded was $24E0.

If one has an object deck to be loaded from another device (other than the scratch LGO device), the same sequence would take place but this time the other logical unit would be used. This would be, for example, an object tape made earlier being loaded. An *T(CR) must follow the last program to be loaded.

There are two forms to the *X(CR) statement. If a nonblank character is used after the X (i.e., X,N(CR)), the memory map (Entry Point Table) will not be printed.

Input



```
Comments                List Device
Device
J
*A
J03,*A
J
*P
J
*L,6
J                       EXAMPL  24E0
*X,N
EXAMPLE PRINTOUT
J
```

Note that when an error on a control statement is made the job processor indicates the type of error, 03, that was made and prints out that error. In the above example, *A was an illegal control statement.

9.1.3 Other Job Processor Control Statements

*K Control Statement

There are times when the programmer would like to change the various standard units. He may prefer that his listing appear on the teletypewriter or be stored on another output device to be printed at a later time. The programmer may change any standard device except the comments deviceby the *K, Iu, Pu, Lu(CR). The I is for the input device with the u standing for the logical unit it is to be changed to. P is for the punch (or binary output) device and the L stands for the list (or print) device. For example, if the programmer wanted to store his relocatable binary program on magnetic tape (logical unit 5 at his installation), his listing to appear on the comments device (logical unit 4), and his source program is on another magnetic tape drive (logical unit 6), he could type the following:

Comments
Device

```
J
*K, P5, L4, I6
J
*ASSEM
OPTIONS XLP
J
```

The parameters P, L, I can be in any order and in any combination. The change will be in effect until another *K statement is given or until the system is autoloaded again. It simply causes the contents of location $F9-$FD to be changed in core.

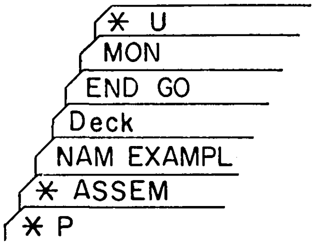
It is usually a good idea for the programmer to refer to standard devices in his program rather than directly to specific logical unit numbers. Then he can reassign the units with the *K control statement if he desires. Particularly, for example, if he was outputting answers on the printer, he could output on the standard print device. Then if the printer was down, the output could be changed to go on the teletype without changing the program.

A standard device is also used by the system when a particular function is to take place but the logical unit number of the particular device to be used may be different for each installation. For example, a message is to be given to the operator but whether the teletype is logical unit 4 or 11, or whether the message is to be printed on a line printer or upon an auxiliary teletypewriter depends upon the purpose and configuration. The actual device to be used can be defined by the logical unit placed in the LOCORE word associated with the standard device. The logical units are assigned at installation time but may always be changed by a *K control statement.

*V and *U

The *V is used to direct the system to expect further control statements from the standard input device. The *U (on the input device) tells the system to switch back to the comments device for control statements.

Input



```
* U
MON
END GO
Deck
NAM EXAMPL
* ASSEM
* P
```

Comments
Device

```
J
*V
J
```

These allow the programmer to put his control statements in the card deck rather than type them on the teletype.

*<Entry Point Name>

When a program has been placed on the program library either during system initialization or through LIBEDT it may be loaded into the system and brought into immediate execution by an *<entry point name>(CR). The assembler and FORTRAN compiler work in this manner as do user written programs.

Comments Device	
J	
*P	
J	
*PGM	Call in PGM
J	

The following is an example of compiling, loading and executing a program under FORTRAN. The TTY printout is shown.

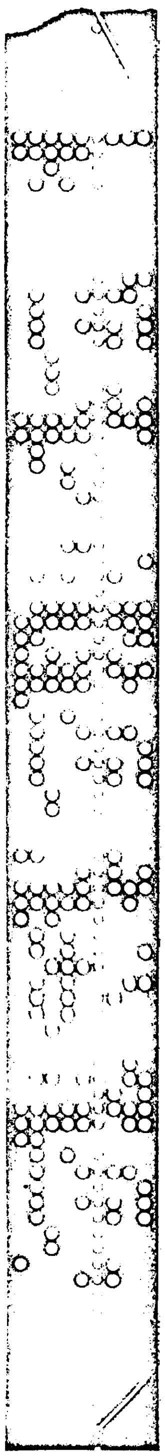
9.1.3

1700 MSOS 2.0 COMPILE AND EXECUTE

Operator must first: STEP - center protect switch - master clear - autoload - run

Circled items are typed by operator

```
PP      <------------------  Set Protect Switch
*       <------------------  Manual Interrupt
MI
*K,I5   <------------------  Assign Input to Card Reader
J
*P      <------------------  Call in Loader
J
*FTN    <------------------  Load FORTRAN
OPTIONS  LAPX  <--------------------  L  list source
           I         I                A  list assembly
        { PROGRAM NAME                P  punch object tape
Source  {    2         2              X  object on disk
 list   { CONTINUE
        {    3         3
        { END
            0000      0000                     NAM      NAME    }
            0000      1801      NAME           JMP*     .00001  }  Assembly list
      3     0001      5400      .00001         RTJ+     Q8STP   }
            0002      7FFF
      3     0000      0000                     END             0

PROGRAM LENGTH $0003

EXTERNALS
Q8STP

J
*P      <------------------  Call in Loader
J
*L,2    <------------------  Load object from PTR
    NAME     2066  <-----  Loaded at $2066
L,02  FAILED  02   <-----  Reader out of tape
ACTION
CU      <------------------  Continue
J
*X      <------------------  Execute             *X,, will eliminate MAP   }
    PSSTOP   2069                                                          }  MAP
    Q8PAND   20A5                                                          }
I ENTRY POINT TABLE-                                                       }
    NAME     2066   Q8STP   2080   Q8PSE    2069   Q8PSEN   206E           }
    Q8STPN   2087   Q8COMI  208A   Q8PAND   20A5                              MAP


STOP
J                            Finished.  Run next job beginning with *P.
```



REL BIN OBJ PT

The next example is one in which the control statements were on the input unit. In this case the card reader was used so the control statements were on cards with the source deck. TTY (comments device) printout and printer (list device) printout are shown.

TTY PRINTOUT CONTROL STATEMENTS ON INPUT UNIT

```
MI
*V       ←—Typed by operator.  "Options" would also be typed by operator.
E *      ←—Typed by operator.
THIS PROGRAM WORKS ON THE 1700 COMPUTER SYSTEM UNDER MSOS 2.0
J                                                    ↗
                                        Program Printout
                                        (Formatted ASCII Write)
```

Deck Setup:

Operator must type *K (if needed to assign units), *V to send control to input unit.

Deck would contain:

```
*P
*ASSEM       (or *FTN)
    NAM               ←———————  If OPT used, operator must type options
                                LAPX, etc.
    END
    NAM

    END      etc.
 MON

*P
*L, 8  ←——— Load from LGO unit (i.e., 8)
*X, ,
       ←——— If missing subroutines, operator must type *CR
*U          To return control to TTY
```

9.1.3

Printer Output
Control Statements on Input Device

```
P      ◄—— *P
ASSEM◄—— *ASSEM
        Deck
  0001                              NAM    CARD TO PRINT
  0002                              ENT    START, PRINT
  0003                              EXT    IOERR
  0004          00EA                EQU    ADISP($EA)
  0005   P0000  0000    START       0      0
  0006   P0001  54F4                RTJ-   ($F4)
  0007   P0002  0201                NUM    $0201          READ, CP=1
  0008   P0003  0009 P              ADC    COMPRD
  0009   P0004  0000                NUM    0, $100C       THREAD, LUN CR=12,    ASCII
         P0005  100C
  0010   P0006  0028                NUM    40             ONE CARD TO READ
  0011   P0007  001F P              ADC    BUF            FWA  BUFFER  AREA
  0012   P0008  14EA                JMP-   (ADISP)
  0013   P0009  0162    COMPRD      SQP    SCHPRT
  0014   P000A  5400  X             RTJ    IOERR
         P000B  7FFF X
  0015   P000C  54F4    SCHPRT        RTJ-    ($F4)
  0016   P000D  1200                NUM    $1200
  0017   P000E  0011  P             ADC    PRINT
  0018   P000F  54F4                RTJ-   ($F4)
  0019   P0010  0A00                NUM    $A00           EXIT REQUEST
  0020   P0011  54F4    PRINT       RTJ-   ($F4)
  0021   P0012  0C01                NUM    $0C01          PRINT,  CP=1
  0022   P0013  001A  P             ADC    COMPPR         COMPLETION ADDRESS
  0023   P0014  0000                NUM    0, $1009, 35
         P0015  1009
         P0016  0023
  0024   P0017  001F  P             ADC    BUF            FWA BUFFER
  0025   P0018  54F4                RTJ-   ($F4)
  0026   P0019  0A00                NUM    $A00
  0027   P001A  0162    COMPPR      SQP    FINI
  0028   P001B  5400 X              RTJ    IOERR
         P001C  000B X
  0029   P001D  54F4    FINI        RTJ-   ($F4)
  0030   P001E  0A00                NUM    $0A00          EXIT WHEN THRU
  0031   P001F  0028    BUF         BSS    BUF(40)
  0032                              END    START

   I           00FF   START     0000P  PRINT    0011P  ADISP    00EA   COMPRD    0009P
   SCHPRT      000CP  COMPPR    001AP  FINI     001DP  BUF      001FP  IOERR     001CX

P          ◄——————— *P

L, 8       ◄——————— *L
    CARD     2210 ◄——Loaded at $2210

X, ,         ◄—————— Execute
  E10
  IOERR      ◄—————— Missing subroutine

U
```

9.2 DEBUGGING

After a program has been written, the programmer will want to run it to determine if the coding meets the specifications that he meant them to satisfy. There are basically two types of errors that may have been made: those of format and those of logic. The assembler and loader detect most of the format errors and notify the programmer via diagnostics.

Two software packages are provided to help the programmer detect errors of logic: the breakpoint package and the system recovery package.

9.2.1 Assembler Errors

If an error is made in the general format of the program the assembler may detect it in either of the first two passes or in the third. If an error is detected during the first two passes, the diagnostic will appear before the NAM card on the listing; otherwise, the diagnostic will be within the body of the program. If at assembly time the L option was chosen, the diagnostics will appear on the list device; otherwise, the diagnostics will appear on the comments device.

When the error is detected during the first or second pass of the assembler, the diagnostic will take the following form:

Column	Contents
1	*
2-5	4-digit card number
6-7	**
8-9	2-character error code
10-19	**********

In the following example a symbol to be used in an IFA pseudo-instruction must be defined before it is used. Therefore, the assembler prints the diagnostic UD, meaning undefined. The 0004 indicates the error occurred in card #4.

LIST DEVICE

```
*0004**UD**********
*0004**UD**********
```

```
0001                      NAM   VARIABLE ASSEMBLY
0002  P0000  5448         ALF   *, THIS WILL ALWAYS ASSEBLE*
      P0001  4953
      P0002  2057
      P0003  494C
      P0004  4C20
      P0005  414C
      P0006  5741
      P0007  5953
      P0008  2041
      P0009  5353
      P000A  4542
      P000B  4C45
0003  P000C  0B00         NOP
*******PP**********
0004                  XX  IFA   DOG, GT, CAT
0005                  ZZ  ALF   *, SOMETIMES ASSEMBLES DEPENDING UPON CAT*
```

When an error is discovered in the third pass of the assembler, the assembler will print the message immediately before the line with the error in the following form:

```
******OP**********
******UD**********
0006    P0006    0000        SHIFT TWO
```

The PP in the first example says the error in that card was identified in the previous pass.

The next example shows the RL error messages for attempted illegal relocation outside the program area (relative). It also shows the EX message for the illegal expression in the address field of the VFD.

```
                         LIST DEVICE
0001                                 NAM      PSEUDO  EX
0002                  0010           EQU      CAT(16)
0003                  0000 D         DAT      BUF(50),EX(50)
                      0032 D
0004                  0000 C         COM      TABLE(50)
0005       P0000      000A           BSS      XX(10),X,Y,Z
           P000A      0001
           P000B      0001
           P000C      0001
0006                  0032 D         ORG      EX
0007       D0032      C400           LDA      TABLE
           D0033      0000 C
0008       D0034      F0F0           NUM      $F0F0,$FFFF
           D0035      FFFF
0009       D0036      0000 D         ADC      BUF
0010                  0000 P         ORG      XX
0011       P0000      5341           ALF      *,SAM IS OK*
           P0001      4D20
           P0002      4953
           P0003      204F
           P0004      4B20
0012                  0000 P         ORG*
0013       P000D      0B00           NOP
0014       P000E      0B00           NOP
************RL**************
0015       P000F      0000  ADDRESS ADC*      BUF,EX,X,Y,Z,ADDRESS
************RL**************
           P0010      0032
           P0011      FFF8
           P0012      FFF8
           P0013      FFF8
           P0014      FFFA
************EX**************
0016       P0015      5800  BYTE     VFD      A8/X,B3/7,B2/1,B8/$FF
```

9.2.2 Device Failure

If while working with the 1700 System there is a device failure, the operating system will print the following:

L,nn Failed ee
ACTION

The nn specifies the logical unit that failed and the ee an error code indicating why it failed.

Error Code	Meaning
00	Input/output hangup (diagnostic timer)
01	Reject (internal or external)
02	Alarm
03	Parity error
04	Checksum error
05	Internal reject
06	External reject

For special errors for each device see the manual for its driver.

The operator may respond in one of the following ways:

RP Repeat the request. This assumes the operator has corrected the condition and wants to complete the operation. The operator may have forgotten to ready the device. Upon receiving a device failure message, he depresses the ready button, types RP and goes on as normal.

CU Indicates the error has not been corrected but the operator would like to continue operating. The program is notified of the error.

DU The device is marked down for this request and all future ones. This allows the operator to get back to the job processor and take the necessary steps to use another device for his program.

CD The same as CU but, also, suspends job processing.

DD The same as DU but, also, suspends job processing.

In the example on the following page the load was from the paper tape reader. The paper tape was read past its end, as there was no *T(CR) on the end of the tape. The CU signalled the loader to continue as all the tapes had been read.

```
J
*L,2
L,02 FAILED 02
ACTION
CU
J
```

If it had been desired to load another tape, the operator would have put it in the reader and typed RP instead.

```
*P
J
*ASSEM
L,02 FAILED 02
ACTION
DU
J
*K, L11, I13
J
*P
J
*ASSEM
OPTIONS XL
J
```

In this case the operator did not want to assemble from the standard device but forgot to change the logical unit before the *ASSEM statement. By marking the device down he got back to the job processor to change devices and continue.

9.2.3 Loading Errors

At the time of loading the loader may detect those types of errors that can only be detected at load time, such as undefined externals, DATA or COMMON declared larger by a second or third program than by the initiating program. The diagnostics appear on the list device preceded by an E.

9.2.4 Logic Errors (Detected During Execution)

There are several standard software packages to help a programmer detect errors of logic. The Breakpoint Package and System Recovery Package aid in debugging programs in the background. UTOPIA and the On-line Debug Package are for debugging in the foreground in the real-time environment. In this chapter the breakpoint package and system recovery package will be discussed.

9.2.4

The breakpoint package allows the programmer to do such things as run his program under different sets of trial data, divide his program into segments and execute only portions of his program at a time or check intermediate results as he goes along. The recovery package only functions after the program has aborted or finished normally. The programmer then can dump core or mass storage to determine the final state of his program and data.

There are several reasons for using a debug package as compared to console debugging. One of the most important on the 1700 is that one can debug while the machine is being shared with other programs. It is also easier as several conversions are made for the programmer and the programmer has a hard copy of his statements and results.

9.2.4.1 The Breakpoint Package

The breakpoint package must be brought in by an *B sometime after an *P and before the program is put into execution. However, the breakpoint control statements are not used until after the *X statement. All numerics in the breakpoint package are in hexadecimal and all addresses are absolute core addresses not program addresses. A BP message from the computer indicates that the breakpoint package is in operation and expects a control statement.

The *B control statement actually causes a flag to be set in the job processor so that after the *X control statement is typed the breakpoint package is brought into core and control is transferred to it. Therefore, the core area the breakpoint package runs in is physically the core area immediately above the area occupied by the background program.

Figure 18. Control Statements Available to Breakpoint Package

Control Statement	Brief Description
*Ahhhh(CR)	Enter register A with the hexadecimal number indicated by the hhhh.
*C(CR)	Continue execution after breakpoint reached
*Daaaa$_1$, aaaa$_2$ (CR)	Dump locations from hexadecimal address aaaa$_1$ through aaaa$_2$.
*Eaaaa, hhhh, ... (CR)	Enter locations in core from hexadecimal address aaaa with data hhhh, hhhh,....
*Ihhhh(CR)	Enter Index I with the hexadecimal number hhhh.
*Jaaaa(CR)	Jump, that is transfer control, to the address given by the aaaa.
*Ms1, w1, s2, w2, n(CR)	Dump data from the mass storage device beginning with the sector and word, s1, w1 through the sector and word indicated by s2, w2. Logical unit n.
*P(CR)	Print the contents of A, Q, I, P and M.
*Qhhhh(CR)	Enter the Q register with the hexadecimal number given by hhhh.
*Raaaa(CR)	Transfer control to and set a breakpoint at location aaaa.
*Saaaa, aaaa, ... (CR)	Set breakpoints at locations indicated by the aaaa's. Maximum of 15 set at one time.
*Taaaa, aaaa, ... (CR)	Terminate breakpoints at the locations specified by the aaaa's.
*T(CR)	Terminate all breakpoints that have been set.
*Z(CR)	Terminates the breakpoint package.

For example:

```
Comments Device                      Notes

*P
J
*ASSEM
OPTIONS XL
J
*K, L4
J
*P
J
*L, 8
      PSEUDO   2544
      PSEUDO   255D
   E8
   QQ                                Note error and overlay
      CLASSP   255D
J
*SR
J
*B
J
*X
1 ENTRY POINT TABLE-
   ***COM   7FCD
   ***DAT   24E0
   QQ       2559     START  2587
BP
```

At this point the programmer can enter any one of the available breakpoint statements.

The programmer may want to break his programs into portions and execute them separately. An Saaaa,aaaa,....CR sets a stop or breakpoint at the addresses ,aaaa, so that during execution when the program reaches this point the program will halt and control will return to the keyboard. The breakpoint program will at that time print the message BP,aaaa indicating that the breakpoint at the address aaaa has been reached and the breakpoint program expects another control statement.

For example, in the program EXAMPL listed as illustration 1 under Section 9.1.1 perhaps the programmer would like to execute his program thru the EXIT on card 5, program location 12. He would probably take the TTY listing which has the address of the first location of his program (24E0). He would then add the program relocatable address of the instruction where he wants the breakpoint to the first location. He would set the breakpoint one instruction after the last instruction that he wants executed. For example:

```
Comments Device                          Notes

J
*P
J
*L,8
     EXAMPL    24E0                 24E0  machine address of program
J                                 +  12  instruction in program
*B                                  24F2  actual address
J
*SR
J
*X
1    ENTRY POINT TABLE-
        GO      24E9
BP
*S24F2
BP
*D24E0,24F2
   24E0   4558  414D  504C  4520  5052  494E  5420  4F55  5420  54F4
   24EA   0C01  24F2  0000  18FC  0009  24E0  54F4  0A00  54F3
BP
*J24E9
EXAMPLE PRINT OUT
BP, 24F2
```

The *S24F2 sets a breakpoint at 24F2. The D24E0, 24F2 dumps that core area. The J24E9 jumps to location 24E9 (P0009) and executes the write. The program stops at 24F2 before executing the SET A, Q instruction.

One can set a maximum of 15 breakpoints with one *Saaaa(CR). However, many more than that can actually be set at any one time. A breakpoint at a location is actually a RTJ to the breakpoint package inserted in place of the actual code. The actual code is kept by the breakpoint package to be executed and to be returned when the breakpoint is removed. Therefore, several considerations should be made when setting a breakpoint. A breakpoint should not be set at a non-executable instruction such as a data word because the program would never get to that word to execute the RTJ, and therefore would not stop at that breakpoint. Also, the breakpoint should not be set at the second word of a two-word instruction because when that instruction is put into execution the RTJ would then be interpreted as an address rather than executed as a jump to the breakpoint package. The breakpoint should not be placed at a location that will be modified or changed during execution. For example, a breakpoint should not be set at an instruction whose address is to be modified or the first word of a subroutine whose entry is via a RTJ for then again the RTJ to the breakpoint package would be modified or destroyed and the result would be unpredictable.

A breakpoint should never be set on an RTJ instruction because the actual instruction is executed in the breakpoint program itself. Hence, the actual program's RTJ would take the wrong address with it.

If a breakpoint is to be cleared, the *Taaaa, aaaa(CR) statement is used. If all the breakpoints that have been set are to be cleared, the *T(CR) statement is used but none of the addresses are specified.

The contents of the registers are printed out with the *P(CR) statement.

```
BP
*P
  REG.  A=18FD    Q=18CD    I=0814    M=9000    P=54FF
BP
```

One can enter each of the registers except M by indicating the register and typing the hexadecimal number that is to be entered.

```
BP
*AF0F0
BP
*IFFFF
BP
*Q1234
BP
*P
  REG.    A=F0F0    Q=1234    I=FFFF    M=9000    P=54FF
BP
```

9.2.4.1

The contents of core can be dumped by the *Daaaa, aaaa statement. The first aaaa specifies the first location the programmer wants to print and the second aaaa specifies the last.

```
BP
*S24F2
BP
*D24E0,24F2
   24E0   4558  414D  504C  4520  5052  494E  5420  4F55  5420  54F4
   24EA   0C01  24F2  0000  18FC  0009  24E0  54F4  0A00  54F3
BP
```

The output from the *P(CR) statement is on the standard list device. In this example the list device has been made the same as the comments device with an *K statement. This example is a dump of the program listed as Illustration 1. Note that the last location printed out (54F3) has been set as a breakpoint.

If only one location is wanted, specify the single address.

```
BP
*D24E0
  24E0    4558
BP
```

If one wants to enter a location in core an *Eaaaa, hhhh, hhhh, ... (CR) is used. The aaaa specifies the first address to be entered and the following hexadecimal numbers to be placed in each sequential location.

```
BP
*E2557,FFFF,0000,FFFF
BP
*D2557,2559
  2557     FFFF     0000     FFFF
BP
*EAAXX
B01,*EAAXX                                    Note the error message
BP
*E2557,AAAA,BBBB,CCCC
BP
*D2557,2559
  2557      AAAA      BBBB      CCCC
BP
```

If a location is to be skipped, i.e., not entered with data, skip that location by typing two commas in a row. This indicates that the location is to be left unaffected. If a location is to be filled with zeros, the zeros must be specified.

When a programmer would like to begin execution of a sequence of programming out of the normal sequence, he may use the jump statement, *Jaaaa(CR) to the instruction to be executed. Execution begins immediately after the *Jaaaa statement. The aaaa is the address of the first instruction to be executed.

```
J
*K, L4
J
*P
J
*L, 8
   EXAMPL    24E0
J
*B
J
*SR
J
*X
1  ENTRY POINT TABLE-
   GO          24E9
BP
*S24F2
BP
*D24E0,24F2
  24E0   4558  414D  504C  4520  5052  494E  5420  4F55  5420  54F4
  24EA   0C01  24F2  0000  18FC  0009  24E0  54F4  0A00  54F3
BP
*J24E9
EXAMPLE PRINT OUT
BP,24F2
```

The *J24E9 caused the jump to P0009 to execute the write.

The return jump *Raaaa is used when an iterative loop is being checked out and the programmer would like a stop at each execution of the loop.

The contents of words on the mass storage device may be dumped using the *Ms1, w1, s2, w2, nCR where:

s1	is the beginning sector number
w1	is the beginning word to be dumped of that sector
s2	is the last sector to be dumped
w2	is the last word of that sector to be dumped
n	is the logical unit of the disk

There are several combinations that one can use. If, while working in the background, the scratch unit is wanted, the n may be omitted and the scratch unit is assumed. If complete sectors are wanted, the word specification can be omitted and the complete sector will print, If one wanted to examine a file that has been stored on sector 24 of the disk, he could do the following:

```
*M24(CR)
```

The programmer may begin in the middle of a sector and dump the rest of the sector by specifying the first sector and first word but omitting the second sector and word.

If the first complete sector of scratch is wanted, type

```
*M(CR)
```

Examples of the *M statement are under the system recovery section as the system recovery's *M works exactly the same as the breakpoint's.

If at any time the program is to be terminated, a MI and an *Z(CR) will do so. An example of *Z being used to terminate a job is in the first program in the system recovery section.

If an error is made while using the breakpoint package, the breakpoint package will print a message beginning with a B. The possible error statements are as follows:

B01, statement — Statement or parameters are unintelligible for the breakpoint program.

B02, hhhh — $hhhh_{16}$ cannot be processed by breakpoint program because it is protected.

B03, hhhh — Breakpoint limit exceeded. $hhhh_{16}$ is the last breakpoint processed.

B04 — Previous *E statement requested entries in protected core. Entries are not processed; breakpoint program waits for new statement.

9.2.4.2 System Recovery Package

The system recovery package is called in with an *SR(CR) before the program is executed just as the breakpoint was. However, the system recovery package does not function and does not accept control statements until after the program has finished normally or aborts. A RE message indicates that Recovery is in and is ready to receive a statement.

Figure 19. Control Statements Available to the System Recovery Package

Control Statements	Brief Description
*Daaaa$_1$, aaaa$_2$(CR)	Dump locations of core beginning with hexadecimal address aaaa$_1$ and ending with hexadecimal address aaaa$_2$.
*Ms1, w1, s2, w2, n(CR)	Dump mass storage unit n from sector and word s1, w1 to sector and word s2, w2.
*T(CR)	Terminate the system recovery package and return to the job processor.
*n(CR)	Change the list device for dumping contents of core or mass storage.

The statements for dumping core and mass storage are the same as for the breakpoint. The output is on the standard list device.

An *T(CR) terminates the system recovery package.

```
J
*P
J
*L,8
    EXAMPL   24E0
J
*B
J
*SR
```

Note that breakpoint and recovery flags may be set

```
J
*X
1 ENTRY POINT TABLE-
    GO          24E9
BP
*J24E9
EXAMPLE PRINT OUT
BP, 24F2
*E24ED,18FB
MI
```

The operator presses the manual interrupt button on the typewriter here if he desires to terminate job execution and enter the Recovery package

```
*Z
RE
*D24E0,24F2
   24E0   4558   414D   5048   4520   5052   494E   5420   4F55   5420   54F4
   24EA   0C01   24F2   0000   18FB   0009   24E0   54F4   0A00   54F3
RE
```

The *D above dumps core from 24E0 through 24F2, after the program has executed.

9.2.4.2

```
BP
*J24E9
EXAMPLE PRINT OUT
RE
*4
RE
*M,15,,12
ERR                      Note:  ERROR Occurred because word 1 is larger than word 2.

RE
*M,15
   SECTOR NUMBER     0000
  0015          5800    0D03    0302    1803    0B00    18F0    5806    0D04    03FB    5803
  001F          0400    1401    0000    0844    E80D    0B00    02FE    A30B    B80B    0104
  0029          0F0A    0121    18DF    18F5    C8E0    1CF2    0000    0181    0039    0019
  0033          0B00    0000    0000    0000    0000    0000    0000    0000    0000    0000
  **

RE
*D2137
  2137          C80D

RE
*D24E0,2558
  24E0          0001    0163    0A00    0001    0002    0003    0004    0005    0006    0007
  24EA          0008    0009    000A    000B    000C    000D    000E    000F    0010    0000
  24F4          0000    FFFE    FFFA    FFFC    FFFB    FFFA    FFF9    FFFF    5443    4520
  24FE          534D    414C    4045    5354    204E    4F2E    2049    5320    4154    204C
  2508          4F43    2E20    0000    24E3    68D5    0842    481D    0C17    5825    54F4
  2512          0D01    0008    0000    18FC    001E    7FEA    54F4    0A00    54F4    0D01
  251C          0008    0000    08FC    0001    255E    54F4    0A00    54F4    0D01    0008
  2526          0000    18FE    000A    0031    0000    0B00    0000    2560    68B3    0C09
  2530          5805    0000    18FD    68F7    18DC    0B00    CCAB    68A8    68A8    D8A8
  253A          0DFE    0161    1CF8    90A4    01A7    0138    CCA1    689E    C800    FF9E
  2544          681A    18F3    0131    18F8    CC99    9897    01A6    012A    CC95    6893
  254E          C893    6810    18E8    0131    18E6    18F8    5448    4520    534D    414C
  2558          4045

RE
*M,124,25,124
   SECTOR NUMBER     0000
  0124          0814
   SECTOR NUMBER     0001
  0001          5803    404F    4144    FFFF    6804    4804    1800    0003    FFFF    FFFF
  000B          5800    00D0    0000    0000    0000    0000    0000    0000    0000    0000
  0015          0000    0000    0000    0000    5000    0000    0000    0000    0000    0000
  001F          2020    0000    0000    0000    0000    0000    0000    0000    0000    0000
  **
   SECTOR NUMBER     0002
  0001          0000    0000    0000    0000    0000    0000    0000    0000    2020    2020
  000B          2020    0000    0000    0000    0000    1400    FFFF    FFFF    FFFF    FFFF
  **
RE
*T                       Note:  ** implies that the 0000 or FFFF continues for the rest of the sector.
J
```

In the previous example the *4 reassigns output to lun4, the TTY. The *M,15 implies sector 0 of scratch from word 15 on. *D2137 dumps location 2137. *D24E0,2558 dumps those locations, inclusive. *M,124,25,124 dumps several consecutive sectors.

PART II

CHAPTER X

CONFIGURING A SYSTEM

10

CHAPTER X - Configuring a System

CONFIGURING A SYSTEM

The content of this chapter will be devoted to the 1700 and its peripherals as a total system. It is designed to assist the presales analyst in configuring a system by considering the interrelationship of the various pieces of hardware. Each has its own characteristics which must relate to and interface with the total hardware configuration.

Figure 20 is a diagram of most of the standard hardware and it should be consulted as a reference from the text in this chapter. Figure 21 also contains the hardware and it includes the new hardware.

10.1 CENTRAL PROCESSOR

The basic 1704 computer consists of the central processor, arithmetic unit, 4K memory and A/Q channel access to the low-speed I/O package via the slow-channel synchronizer, equipment number 1.

Memory modules may be added in 4K (1708) increments to a maximum of 32K. The 1709 (8K module) is available only on a used basis. A new hardware addition has recently been made to the product line to allow an increase of memory size to 65K.

Two interrupt lines are included: line 0 for internal interrupts and line 1 for the slow-channel synchronizer.

10.1.1 Low Speed I/O Package

The low-speed package consists of:

- teletypewriter

  1711 - keyboard entry and printer only, 100 characters per second

  1712*- keyboard & printer, 100 cps, with offline mechanical paper tape reader and punch

  1713 - keyboard, printer, on-line mechanical paper tape reader and punch, 100 cps

- paper tape reader

  1721 - 400 cps reader, electronic

  1722 - same reader, with added take-up and supply reels

- paper tape punch

  1723 - 120 cps punch

  1724 - same punch, with added take-up and supply reels

- card reader

  1729*- 100 card-per-minute reader, replaced by 1729-2 which connects to 1705

*Only available used on an as-available basis.

10.1.1

These are the standard peripherals connected to line 1; other peripherals must normally be connected through a 1705. Some existing configurations do have special peripherals other than those above on line 1, but this is on a QSE basis and each case must be considered individually.

10.2 1705 INTERRUPT DATA CHANNEL

Any system which will require more than the two basic interrupts or the peripherals in the low-speed I/O package will need a 1705. The addition of the 1705 will add the following capabilities to the system:

a) addition of 14 more interrupt lines for external equipment

b) addition of up to eight controllers to the A/Q channel

c) addition of direct memory access for up to eight buffered controllers

See note A, Figure 20, for additional explanation of line connections.

Input/Output for the additional eight controllers would be unbuffered if they are connected to the A/Q channel only. I/O will be buffered if they are connected to the A/Q channel and the DSA bus and are either capable of doing direct memory access on their own or are connected through a 1706 or 1716.

10.3 BUFFERED CONTROLLERS

Three controllers are capable of performing data transfers directly between computer memory and the attached peripheral device:

1738 disk controller - controls one or two 853 (1.5-million-word) or 854 (3.0-million-word) disks,

1751 drum controller - controls drum; size from 65K to 524K words,

1748 master communications terminal controller - controls up to 64 remote communications sets (through 8136's) or up to four 302 communications expansion modules.

The buffered controllers are connected to the A/Q channel (for transfer of control information) and the DSA bus (for transfer of data). Any program which is running and using the CPU continues to run while the controller is handling the data transfer, since the A and Q registers are not used during the data transfer.

Direct memory transfer is done on a cycle-stealing basis; that is, the controller steals a memory cycle every time it wishes to transfer a word (through the Z register). In order to calculate how much each buffered controller will slow down the CPU (and, therefore, a running program), a percentage could be figured based on the transfer rate of the peripheral. For example, the 1738 is capable of transferring one word every 12.8 μs so it will steal approximately every 12th cycle and can therefore slow down a program by up to 9% when the disk is running. The program would only be slowed down if the CPU and the 1738 both wanted to access memory at the same time.

10.4 1706 BUFFERED DATA CHANNEL

The 1706 allows direct memory access for unbuffered controllers. It is for any of the controllers that are not capable of doing buffered data transfers. (That is, any except the 1738, 1751 and 1748.) The 1706 is connected between the 1705 (DSA and A/Q channels) and the peripheral controller.

Only three 1706's are allowed in any system; this is a software limitation rather than a hardware limitation. The 1716 is exactly like a 1706 except it is accessible by two computers. See note C of Figure 20. The 1706 may control up to eight controllers. However, when deciding which peripherals should go on the 1706, it is very important to note that it is logically busy the entire time it is handling a buffer transfer for a peripheral. During that time it cannot be accessed to do any operation or take status on any other peripheral connected to it. Therefore, the timing on the peripherals must be considered so that data will not be lost on one while the 1706 is working on another.

As a rule, a 1706 would not be purchased to handle relatively slow peripherals (i.e., the 1742 line printer or 430 card reader/punch). These peripherals can very effectively be operated in interrupt mode as they will interrupt the CPU infrequently to perform their I/O. The 1706 would more effectively be used and needed to handle fast peripherals (such as magnetic tapes or the 405 card reader). For example, a 1732/608 magnetic tape can transfer one frame of data every 32 µs. Since MSOS can lock out interrupts for up to 50 µs at one time, data could be lost on the tapes if they were not connected to a 1706.

Software for controllers operated in the buffered mode through the 1706 should be considered on an individual basis in the light of new software releases.

10.5 UNBUFFERED CONTROLLERS

The standard controllers which could be operated in the unbuffered mode through the 1705 to the A/Q channel are as follows. Most are shown in Figure 20. The newer ones are in Figure 21.

- 1726/405 Card Reader - 1200 cpm reader. 405 can also be connected through 1750 via a 177 controller. Shown on Figure 21.
- 1728/430 Card Reader/Punch - 500 cards per minute read; 100 cards per minute punch, (column punch). Reader can be purchased separately as a 1729-2, 330-cpm. Punch can be purchased separately as a 420A, 100 cpm.
- 1729-2 Card Reader - this is the replacement for the 1729 and it reads 330 cpm. Not shown on Figures.
- 1731/601 Magnetic Tapes - this is the 1x8 controller for 601 magnetic tapes (200, 556 bpi). They do not have assembly/disassembly mode. They have been updated by the newer 1732/608-609 tapes, and are now available only on a used basis.

- 1732/608-609 Magnetic Tapes - the new controller, featuring assembly/disassembly option, which replaces the 1731. One controller can handle up to eight tape units; 608's or 609's or a combination of both.

  608's - 7 track; bcd or binary; 200, 556, 800 bpi; read forward and reverse
  609's - 9 track; binary only; 800 bpi only; read forward and reverse

  Software from the 1731/601 is completely upward compatible with 1732 hardware.

  The 1732 is a more expensive controller than the 1731, but the added features would be desirable for the more sophisticated user:

  assembly/disassembly mode*
  800 bpi
  forward and reverse read
  9-track tape

- 1735/915 Page Reader - optical character recognition equipment, 370 characters per second. Software operates as a compiler under Utility System; no standard driver is as yet available under MSOS, but it is planned to be added.

- 1736-1 OCR Document Reader Controller - controls one 935-1 or 935-2 document reader. Software not yet available; will probably run under Utility first.

- 1740/501-505 Line Printers - 501, 1000-lpm printer or 505, 500-lpm; 136 characters. Software not yet available for 505.

- 1742 Line Printer (with controller included) - this is the Holley 300-lpm with control, 136 columns.

- 1744 Digigraphics Controller - controls one 274 digigraphic light-pen console. Shown on Figure 21. Software is QSS.

- 1745-1 Inquiry/Retrieval Controller - controls 211 Display/Entry and 218 Output stations.

- 1746-1 Single Station Entry/Display - controls CRT display and keyboard.

- 1747 Data Set Controller - controls 301-B data sets. Software runs under Utility System. Standard software is available for 6000 import/export.

- 1749 Communications Terminal Controller - controls remote communications equipment, up to 16 lines per controller. Standard software is available only in the unbuffered mode on the 1749; software is not available to connect the 1749 through the 1706 in a buffered mode. The 1748 is used for buffered operations.

Standard software is available now or will be shortly on most of the peripherals above (except as noted) to run under MSOS in the unbuffered mode.

*Also, new software for the 1732 utilizing assembly mode will mean the tapes only have to be accessed half as often.

10.6 160-A PERIPHERALS

Several 160-A peripherals are connected to the 1700 on existing configurations through a 160-A adapter, and the 1750:

a) 405 card reader (through 177 controller)
b) 166 line printer and control
c) 415 card punch (through 170 controller)
d) 165-2 Calcomp plotter and control

These equipments are not listed as standard available products as they are only available used on an as-available basis.

10.7 1500 EQUIPMENT

The 1500 series of analog equipment for process control is all connected to the 1700 through different interfaces. A large, detailed chart of 1500 equipment is available through ADSD in La Jolla. Much of the series is shown in the Figure 20 chart and its primary interfaces are:

a) 1750 DCB Terminator - this is the prime interface and it allows A/Q channel access to the 1500 series via the 1705. It is required if any 1500 equipment is to be connected to the computer.

b) 1797 Buffered I/O Interface - this provides access to the DSA bus for buffered 1500 equipment. It is functionally equivalent to the 1706 for standard peripherals. It is connected to the DSA and the 1750 and it controls up to three 1571's.

c) 1571 Chaining Buffer Channel - this is the priority buffer channel which assigns priorities to the equipment on the 1797. It is required if a 1797 is present. A high priority piece of equipment can steal the channel away from a low priority equipment.

See notes E-J of Figure 20.

One piece of 1500 equipment will be mentioned here as it is nearly always needed in all configurations: 1573 Line Synchronized Timing Option.

This is the clock connected to the 1750 which generates timed interrupts to the CPU (60/sec). Any process system which requires a clock for timed programs will need a 1573 since there is no realtime clock in the computer itself. Many standard systems need a clock, especially to monitor I/O which can get hung up (for example, the 1706 can hang up in a buffer operation if the peripheral malfunctions or drops Ready). Several controllers are capable of generating the necessary timed interrupt, but if one of these is not present in the system, the 1573 can be used.

10.8 PRIORITIES FOR DSA BUS

Since all memory accesses, even buffered, must go through the Z register, priorities must be assigned to all the interfaces which may use the DSA bus. The 1797 takes

10.8

highest priority in direct memory access, the 1706 takes second priority, the standard buffered equipments (1738, 1751, 1748) take third priority. The running program takes lowest priority for accessing memory. On Figure 20, see ①, ②, and ③.

10.9 SUMMARY, CONFIGURING EQUIPMENT

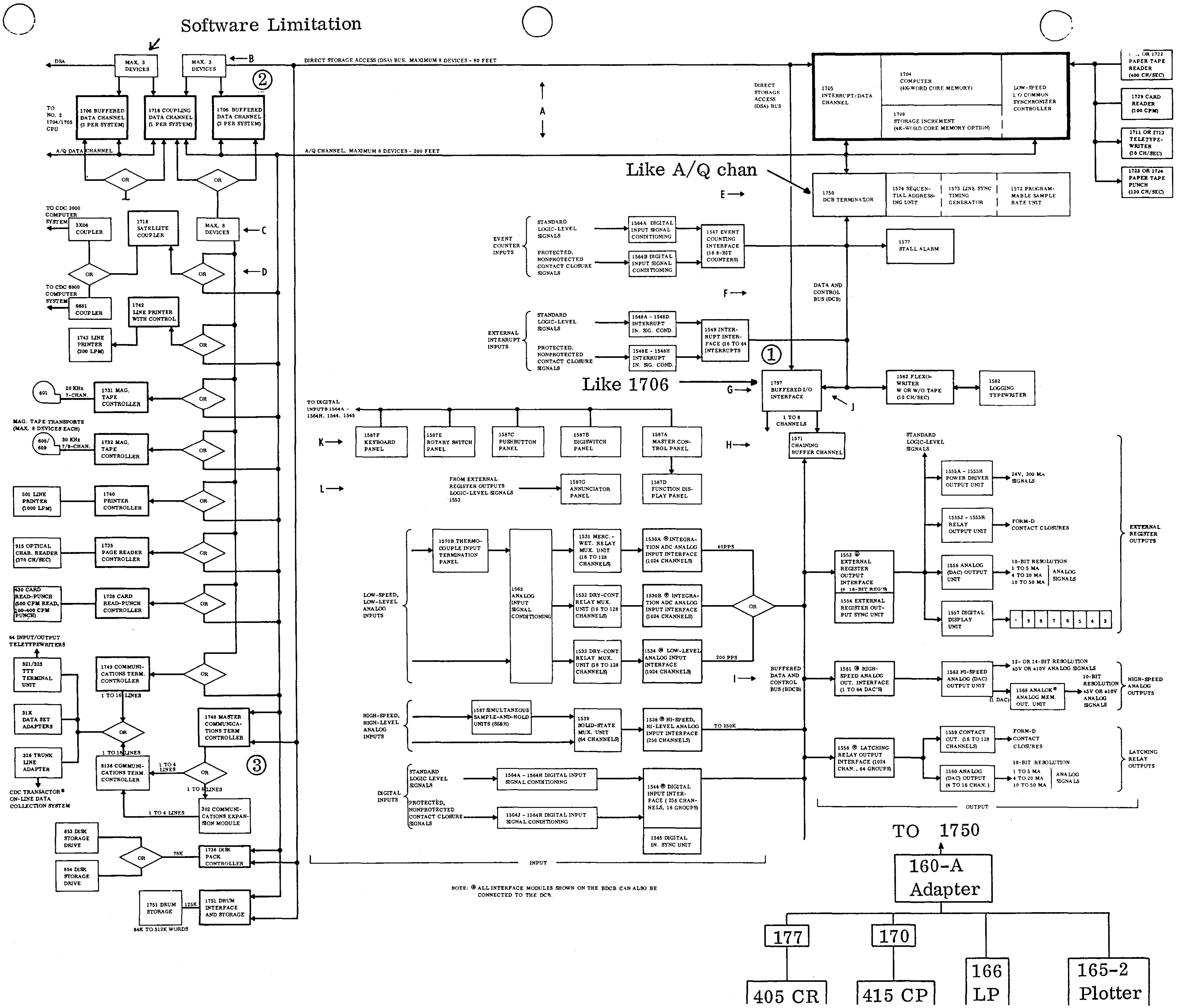
The chart in Figure 20 can be utilized very effectively to configure a system. Note that low-speed I/O package is connected to the CPU through the Low-Speed I/O Synchronizer. Each of the standard buffered equipments (1738, 1751 and 1748) has line connections leading to the CPU via the A/Q channel line and the DSA line. The unbuffered equipments connect to either the A/Q line or, through a 1706, to the DSA line and A/Q line (to add buffer capability).

The 1750 connects to the 1705 (for A/Q access). The 1797 has lines to the 1750 and DSA, and the 1571 leads directly to the 1797. Note that all 1500 peripherals connect either to the 1571, 1797 or 1750. The 160-A peripherals connect through the 1750.

10.10 RELATED MANUALS

Additional information on systems configuration will be found in:

Systems Manual
Communications Peripheral Equipment Manuals
ADSD General Information Manual
Pricing Manual



Software Limitation
DSA
MAX. 3 DEVICES
MAX. 3 DEVICES
B
DIRECT STORAGE ACCESS (DSA) BUS, MAXIMUM 8 DEVICES - 80 FEET
②
TO NO. 2 1704/1705 CPU
1706 BUFFERED DATA CHANNEL (3 PER SYSTEM)
1716 COUPLING DATA CHANNEL (1 PER SYSTEM)
1706 BUFFERED DATA CHANNEL (3 PER SYSTEM)
A/Q DATA CHANNEL
A/Q CHANNEL, MAXIMUM 8 DEVICES - 200 FEET
A
OR
OR
TO CDC 3000 COMPUTER SYSTEM
3X06 COUPLER
1718 SATELLITE COUPLER
MAX. 8 DEVICES
C
D
OR
OR
TO CDC 6000 COMPUTER SYSTEM
6681 COUPLER
1742 LINE PRINTER WITH CONTROL
1742 LINE PRINTER (300 LPM)
OR
601
20 KHz 7-CHAN.
1731 MAG. TAPE CONTROLLER
OR
MAG. TAPE TRANSPORTS (MAX. 8 DEVICES EACH)
608/609
30 KHz 7/9-CHAN.
1732 MAG. TAPE CONTROLLER
OR
501 LINE PRINTER (1000 LPM)
1740 PRINTER CONTROLLER
OR
915 OPTICAL CHAR. READER (370 CH/SEC)
1735 PAGE READER CONTROLLER
OR
430 CARD READ-PUNCH (500 CPM READ, 100-400 CPM PUNCH)
1728 CARD READ-PUNCH CONTROLLER
OR
64 INPUT/OUTPUT TELETYPEWRITERS
321/323 TTY TERMINAL UNIT
1749 COMMUNICATIONS TERM. CONTROLLER
OR
1 TO 16 LINES
31X DATA SET ADAPTERS
OR
326 TRUNK LINE ADAPTER
CDC TRANSACTOR® ON-LINE DATA COLLECTION SYSTEM
1 TO 16 LINES
8136 COMMUNICATIONS TERM. CONTROLLER
1 TO 4 LINES
1748 MASTER COMMUNICATIONS TERM CONTROLLER
OR
③
1 TO 8 LINES
1 TO 4 LINES
302 COMMUNICATIONS EXPANSION MODULE
853 DISK STORAGE DRIVE
OR
78K
1738 DISK PACK CONTROLLER
854 DISK STORAGE DRIVE
1751 DRUM STORAGE
125K
1751 DRUM INTERFACE AND STORAGE
64K TO 512K WORDS
DIRECT STORAGE ACCESS (DSA) BUS
1705 INTERRUPT/DATA CHANNEL
1704 COMPUTER (4K-WORD CORE MEMORY)
1708 STORAGE INCREMENT (4K-WORD CORE MEMORY OPTION)
LOW-SPEED I/O COMMON SYNCHRONIZER CONTROLLER
1721 OR 1722 PAPER TAPE READER (400 CH/SEC)
1729 CARD READER (100 CPM)
1711 OR 1713 TELETYPEWRITER (10 CH/SEC)
1723 OR 1724 PAPER TAPE PUNCH (120 CH/SEC)
Like A/Q chan
E
1750 DCB TERMINATOR
1574 SEQUENTIAL ADDRESSING UNIT
1573 LINE SYNC TIMING GENERATOR
1572 PROGRAMMABLE SAMPLE RATE UNIT
EVENT COUNTER INPUTS
STANDARD LOGIC-LEVEL SIGNALS
PROTECTED, NONPROTECTED CONTACT CLOSURE SIGNALS
1564A DIGITAL INPUT SIGNAL CONDITIONING
1564B DIGITAL INPUT SIGNAL CONDITIONING
1547 EVENT COUNTING INTERFACE (16 8-BIT COUNTERS)
1577 STALL ALARM
DATA AND CONTROL BUS (DCB)
F
EXTERNAL INTERRUPT INPUTS
STANDARD LOGIC-LEVEL SIGNALS
PROTECTED, NONPROTECTED CONTACT CLOSURE SIGNALS
1548A - 1548D INTERRUPT IN. SIG. COND.
1548E - 1548H INTERRUPT IN. SIG. COND.
1549 INTERRUPT INTERFACE (16 TO 64 INTERRUPTS)
①
Like 1706
G
1797 BUFFERED I/O INTERFACE
J
1582 FLEXOWRITER W OR W/O TAPE (10 CH/SEC)
1582 LOGGING TYPEWRITER
TO DIGITAL INPUTS 1564A - 1564H, 1544, 1545
K
1587F KEYBOARD PANEL
1587E ROTARY SWITCH PANEL
1587C PUSHBUTTON PANEL
1587B DIGISWITCH PANEL
1587A MASTER CONTROL PANEL
H
1 TO 8 CHANNELS
1571 CHAINING BUFFER CHANNEL
L
FROM EXTERNAL REGISTER OUTPUTS LOGIC-LEVEL SIGNALS 1553
1587G ANNUNCIATOR PANEL
1587D FUNCTION DISPLAY PANEL
STANDARD LOGIC-LEVEL SIGNALS
1555A - 1555H POWER DRIVER OUTPUT UNIT
24V, 300 MA SIGNALS
1555J - 1555R RELAY OUTPUT UNIT
FORM-D CONTACT CLOSURES
EXTERNAL REGISTER OUTPUTS
LOW-SPEED, LOW-LEVEL ANALOG INPUTS
1570B THERMOCOUPLE INPUT TERMINATION PANEL
1563 ANALOG INPUT SIGNAL CONDITIONING
1531 MERC.-WET. RELAY MUX. UNIT (16 TO 128 CHANNELS)
1530A ® INTEGRATION ADC ANALOG INPUT INTERFACE (1024 CHANNELS)
40PPS
1532 DRY-CONT. RELAY MUX. UNIT (16 TO 128 CHANNELS)
1530B ® INTEGRATION ADC ANALOG INPUT INTERFACE (1024 CHANNELS)
OR
1533 DRY-CONT. RELAY MUX. UNIT (16 TO 128 CHANNELS)
1534 ® LOW-LEVEL ANALOG INPUT INTERFACE (1024 CHANNELS)
200 PPS
1553 ® EXTERNAL REGISTER OUTPUT INTERFACE (8 16-BIT REG'S)
1554 EXTERNAL REGISTER OUTPUT SYNC UNIT
1556 ANALOG (DAC) OUTPUT UNIT
10-BIT RESOLUTION 1 TO 5 MA 4 TO 20 MA 10 TO 50 MA
ANALOG SIGNALS
1557 DIGITAL DISPLAY UNIT
- 9 8 7 6 5 4 3
I
BUFFERED DATA AND CONTROL BUS (BDCB)
1561 ® HIGH-SPEED ANALOG OUT. INTERFACE (1 TO 64 DAC'S)
1562 HI-SPEED ANALOG (DAC) OUTPUT UNIT
12- OR 14-BIT RESOLUTION ±5V OR ±10V ANALOG SIGNALS
(1 DAC)
1568 ANALOK® ANALOG MEM. OUT. UNIT
10-BIT RESOLUTION ±5V OR ±10V ANALOG SIGNALS
HIGH-SPEED ANALOG OUTPUTS
HIGH-SPEED, HIGH-LEVEL ANALOG INPUTS
1567 SIMULTANEOUS SAMPLE-AND-HOLD UNITS (SS&H)
1539 SOLID-STATE MUX. UNIT (64 CHANNELS)
1538 ® HI-SPEED, HI-LEVEL ANALOG INPUT INTERFACE (256 CHANNELS)
TO 250K
1558 ® LATCHING RELAY OUTPUT INTERFACE (1024 CHAN., 64 GROUPS)
1559 CONTACT OUT. (16 TO 128 CHANNELS)
FORM-D CONTACT CLOSURES
LATCHING RELAY OUTPUTS
1560 ANALOG (DAC) OUTPUT (4 TO 16 CHAN.)
10-BIT RESOLUTION 1 TO 5 MA 4 TO 20 MA 10 TO 50 MA
ANALOG SIGNALS
DIGITAL INPUTS
STANDARD LOGIC LEVEL SIGNALS
PROTECTED, NONPROTECTED CONTACT CLOSURE SIGNALS
1564A - 1564H DIGITAL INPUT SIGNAL CONDITIONING
1564J - 1564R DIGITAL INPUT SIGNAL CONDITIONING
1544 ® DIGITAL INPUT INTERFACE (256 CHANNELS, 16 GROUPS)
1545 DIGITAL IN. SYNC UNIT
OUTPUT
INPUT
NOTE: ® ALL INTERFACE MODULES SHOWN ON THE BDCB CAN ALSO BE CONNECTED TO THE DCB.
TO 1750
160-A Adapter
177
170
166 LP
165-2 Plotter
405 CR
415 CP

Figure 20. 1700 Computer System Block Diagram

Figure 20 (cont)

Figure 20 is an overall view of a CONTROL DATA® 1700 Computer System; it shows how different subsystems are used and the methods by which they can be connected. A few basic facts about the CDC® 1700 Computer System are pointed out below. Each statement corresponds to letters on the block diagram.

A Up to eight input/output (I/O) controllers can be connected directly to the A/Q channel and up to eight I/O controllers can be connected directly to the direct storage access (DSA) bus. However, this does not mean that 16 I/O controllers can be directly connected; only eight can be connected because each controller that is connected to the DSA bus must also be connected to the A/Q channel.

B Only three CDC 1706 Buffered Data Channels and/or the buffered 1716 Coupling Data Channels can be used in a system complex; this is an addressing restriction, not a hardware restriction.

C Each Model 1706 or 1716 provides up to eight data channels to which I/O subsystems can be connected. Either the 1706 or the 1716 permits the attached subsystems to operate via the direct storage access (DSA) bus.

D The "OR" box indicates that the attached I/O subsystem can either operate through the 1716 or connect directly to the A/Q channel. Operation via the 1706 is in the buffered mode; operation via the A/Q channel is in the unbuffered mode.

E The CDC Model 1750 DCB Terminator is required whenever a CDC 1500 Series subsystem is used with the CONTROL DATA 1700 Computer. The 1750 provides a data and control bus (DCB) which is functionally equivalent to the A/Q channel.

F The DCB provides the capability of attaching up to 15 CDC 1500 Series I/O subsystems, all of which operate through the A/Q channel via the Model 1750.

G The CDC 1797 Buffered I/O Interface is required when the attached CDC 1500 Series subsystems must operate through the DSA bus; it provides up to eight priority buffer channels to which the CDC 1500 Series equipment can be connected.

H The CDC 1571 Chaining Buffer Channel connects to one of the eight priority buffer channels of the Model 1797. Up to three 1571's can be connected to one 1797. Each 1571 uses one priority buffer channel.

NOTE

If a 1797 is included in a system, the Model 1571 is also required to connect existing CDC 1500 Series I/O subsystems.

I Each 1571 provides a buffered data and control bus (BDCB) to which up to 15 CDC 1500 Series I/O subsystems can be connected.

All devices that are shown connected to the BDCB can be connected to the DCB (refer to F, above); this means that the 1797/1571 is only required by system definition.

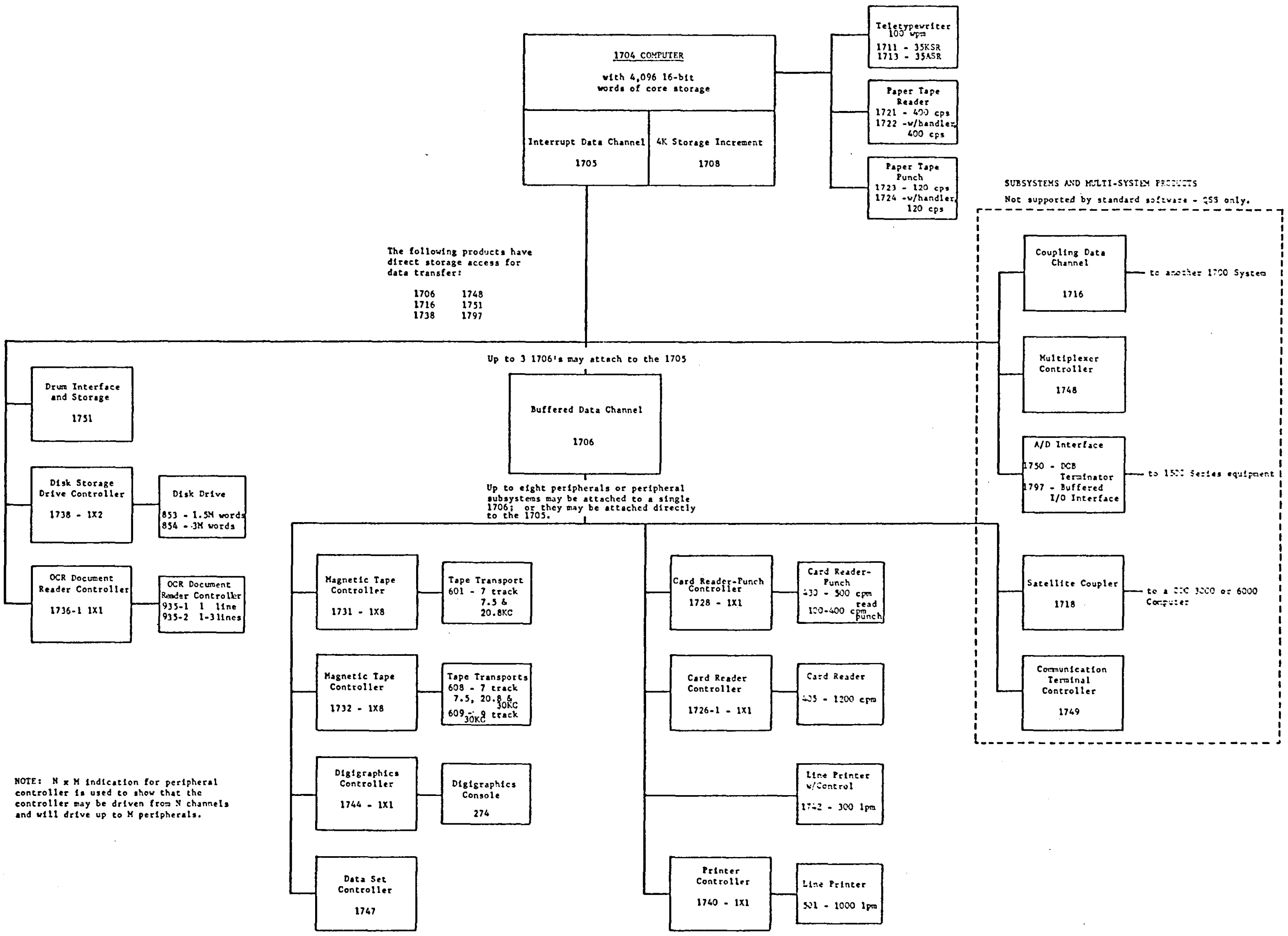
Figure 20 (cont)

J Subsystems connected to the BDCB normally operate through the Model 1797 and the DSA bus. However, by program control, they can operate through the Model 1750 and the A/Q channel.

K The 1587A Master Control Panel, 1587B Digiswitch Panel, 1587C Pushbutton Panel, 1587E Rotary Switch Panel and 1587F Keyboard Panel connect to the 1564A through 1564H Digital Input Signal Conditioning and operate through the 1544 Digital Input Interface with the 1545 Digital Input Sync Unit.

L The 1587G Annunciator Panel receives its information directly from a 1553 External Register Output Interface.

1700 HARDWARE CONFIGURATION



The following products have direct storage access for data transfer:

1706	1748
1716	1751
1738	1797

Up to 3 1706's may attach to the 1705

Up to eight peripherals or peripheral subsystems may be attached to a single 1706; or they may be attached directly to the 1705.

SUBSYSTEMS AND MULTI-SYSTEM PRODUCTS

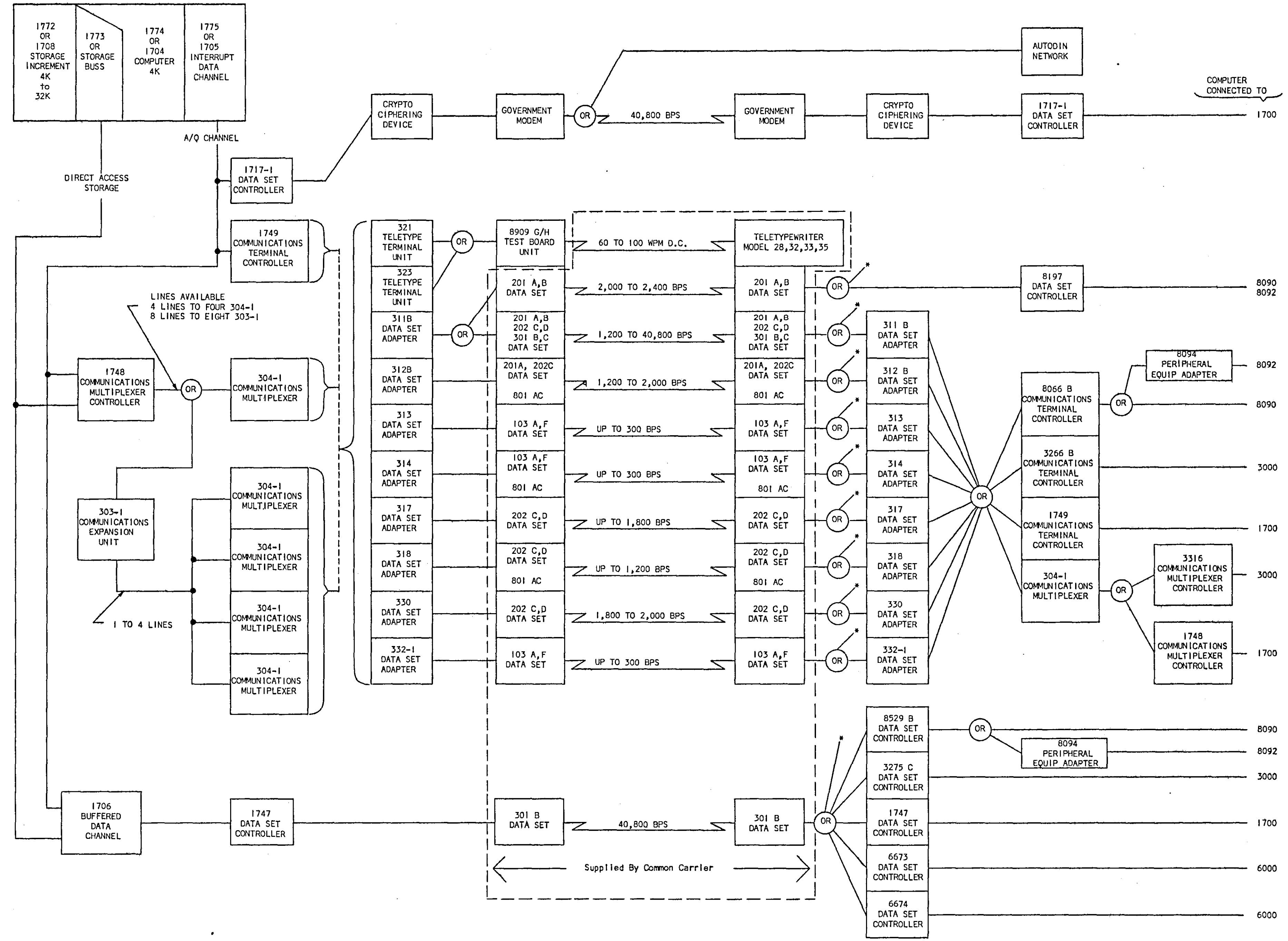
Not supported by standard software - QSS only.

NOTE: N x M indication for peripheral controller is used to show that the controller may be driven from N channels and will drive up to M peripherals.

* Standard software support for the above four products is available only when they are attached to the 1705.

Figure 21. 1700 Hardware Configuration

Figure 22 shows the latest configuration of communications equipment through the various communications controllers.



*Indicates CDC terminal devices using compatible data sets may be used. Use of other than CDC devices must be coordinated with CSD Product Management.

Figure 22. 1700 (1704 or 1774) Communications System Configuration

The following is a systems bulletin describing buffered and non-buffered operations:

BUFFERED/NON-BUFFERED OPERATIONS

The purpose of this Data Sheet is to define the terms "Buffered and Non-Buffered" operations in terms of hardware. Hopefully, this will eliminate any misconceptions in actual hardware operations relative to the terms. It should be made clear that all I/O devices used with the 1700 Computer are buffered in regard to the handling of data. Each output device or subsystem has a buffer into which the computer can load data. This is a temporary storage media that holds the data while the output device goes through its slow-speed operation using that data. During this time, the computer is free to continue on with its program. Each input device or subsystem contains a buffer media into which it loads data until the running program can accept it as input and during which time the input device is obtaining the next set of data for entry into the buffer.

Non-Buffered Operations

The term "Non-Buffered" operation is synonomous with "Direct" operation, and means that an I/O operation is in progress and data is being transferred during the execution of an I/O instruction via the computer's A/Q Channel. Therefore, data is either being input to the A-Register or output from the A-register. The Q-register, in each case, holds an address specifying the equipment or device from which the data is coming or to which the data is going. Output data goes to a buffer for temporary storage until the device can use it and input data comes from a buffer where it has been waiting. This is the Non-Buffered or Direct I/O Operation.

Buffered Operations

The term "Buffered" operation means that the program has initiated an I/O operation for the Direct Storage Access (DSA) bus and is then free to continue its program while the actual data transfer is completed. The running program is not interrupted until the entire record has been either read or written. Typically, a buffered operation is carried out as follows:

1) Normally initiated by the computer program executing a "Non-Buffered" input or output instruction to a device connected to both the A/Q channel and DSA bus. This Non-Buffered output would be the instruction telling the device or subsystem to start operating in the "Buffered" mode.

2) Information supplied to the device during this Non-Buffered operation would be a "Pointer Word" that would point to a set of control words located in memory.

3) These control words may represent the Starting and Ending memory addresses (Record length) plus any other information required by the device. They may also represent the starting and ending addresses used by the I/O device.

4) Once the Non-Buffered operation is completed, the program is free to continue its function.

5) When the specified device wants data or has data available, it requests a memory cycle.

6) On the next available memory cycle, the data word is transferred to or fetched from the memory. Therefore, the program is delayed for one memory cycle each time a data word is transferred in or out of memory. This is commonly referred to as cycle stealing.

7) After each data transfer, the device or subsystem increments the current address and compares it with the Ending address. Steps 5, 6 and 7 are repeated until the current address and Ending address match (held in the 1571 or its equivalent).

8) When the entire record has been transferred, the Buffered operation stops and the program can be interrupted.

Direct Storage Access (DSA)

Direct Storage Access is commonly referred to as Direct Memory Access or may just be called a memory channel. The 1700 Computer DSA contains a Data Register, Memory Address Register and control logic which enables attached devices or subsystems to request memory cycles and receive access to the memory on a priority basis. The DSA bus always have a higher priority than the arithmetic unit. With no Memory request from DSA, the arithmetic units requests will continue to be granted for consecutive memory cycles. However, a Memory request from the DSA has first priority so that when received, the next memory cycle is granted for DSA use. Once the DSA has control of the memory, it would use as many memory cycles as necessary to satisfy all requests from the devices on the DSA bus. For example, if six devices were attached to the DSA bus and all requested memory at the same time, the program would be effectively stalled for six memory cycles (6.6 μs) while these six requests were serviced.

Summary

In summary, a Non-Buffered operation passes data in and out of the A-register via the A/Q channel while the Buffered operation passes data in and out of the memory on the Direct Storage Access Bus.

CHAPTER XI

ADVANCED CODING TECHNIQUES

CHAPTER XI - Advanced Coding Techniques

11.0 INTRODUCTION

Advanced coding techniques for the 1700 will be the special considerations needed for understanding and writing programs that are part of the system. There are two separate libraries of programs: the Program Library and the System Library.

The programs in the Program Library are relocatable binary programs which are run in the background as jobs. This would include such programs as the library subroutines needed by the jobs. They are loaded into the unprotected background area of core by the loader, for execution, after being called in from the teletypewriter. Jobs could also be loaded from the card reader or paper tape reader for execution.

The programs in the System Library are absolute programs which are part of the system; they are run in the foreground in a large area of protected core called allocatable core. This would include user process programs. The Operating System contains a directory of all the system programs (all the mass memory programs and maybe a few core resident programs). This is like a list of all the programs, by module name, and it contains the addresses of where they are. The system uses the directory to find the programs when it is desired to bring them into core to execute them.

Protected core is normally synonymous with the foreground, and unprotected core with the background. In the background only one program (and its subroutines) would be in execution at one time at the lowest priority. In the foreground many programs could be in various states of execution at different priorities. The lower priority ones might have been suspended (temporarily stopped) while the highest priority one is executed. The unfinished programs wait in core to finish execution. If the system needs more allocatable core because it is full, it will "swap out" the entire background area to the Swap area on mass memory, protect the background area, and use it for foreground programs. When enough foreground programs are completed in order to release the background area, that core is unprotected again and the job swapped back in to continue.

Note that some of the system programs are core resident. The reason all of them are not core resident is that they will not all fit in core at once. Therefore, the ones which are not needed all the time reside on mass memory in the System Library and are called into allocatable core as they are needed.

In order to understand what all these programs look like and how they execute, as well as how they call each other in to core, it will be necessary to study the different kinds of programs.

Emphasis in this chapter will be on system programs as background programs will run with any of the coding techniques previously covered.

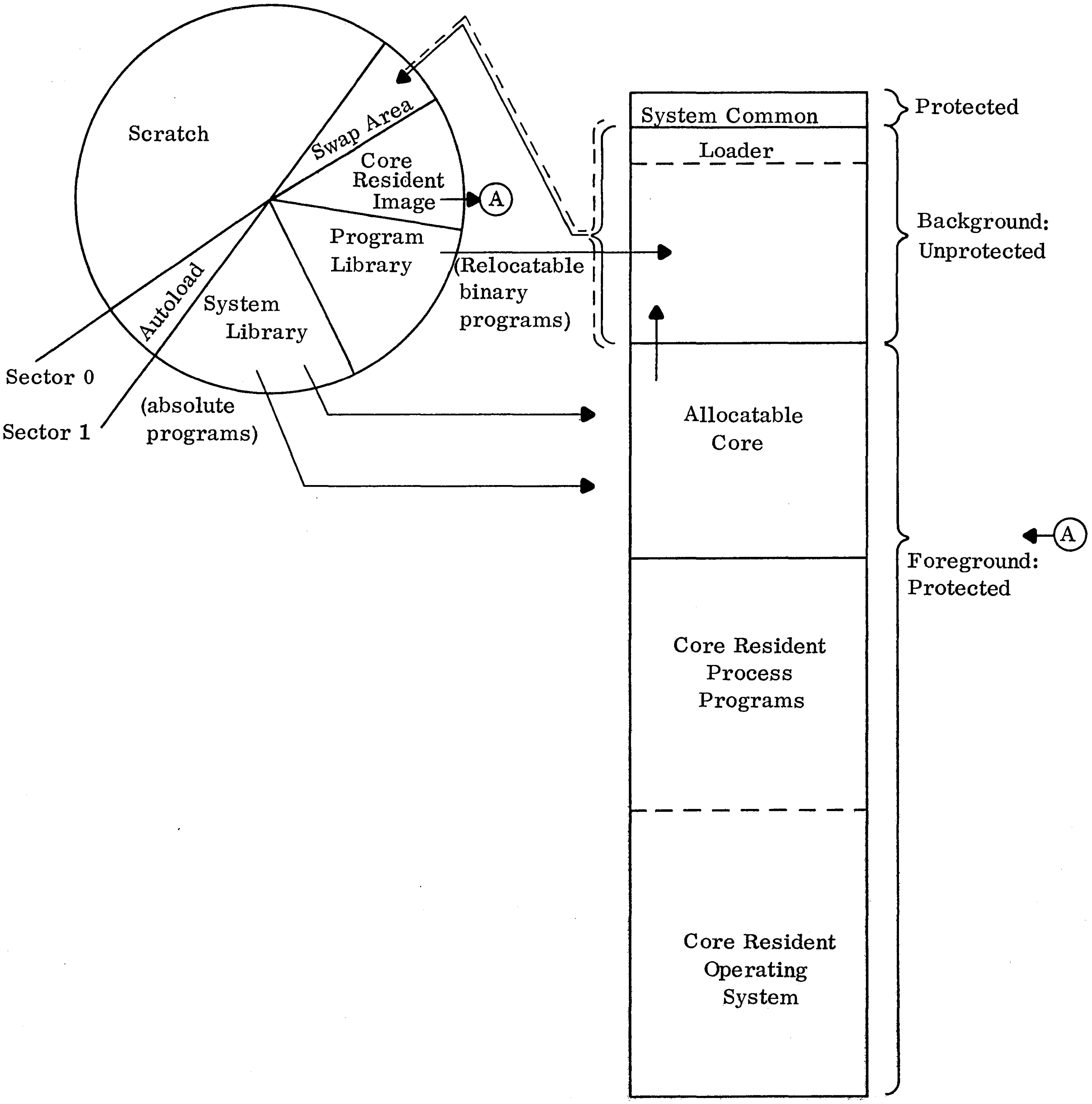


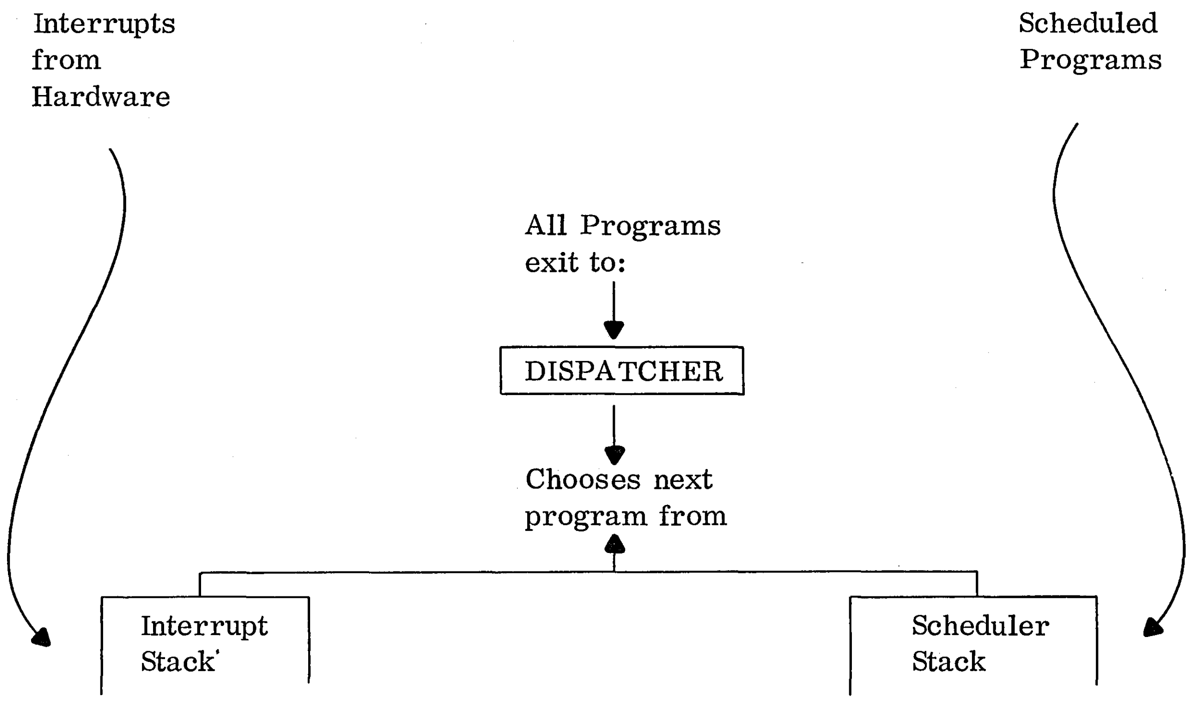
Figure 23. Mass Memory and Core Maps

An introduction to the priority structure of the running programs under MSOS will also be helpful as an introduction to this chapter. There are 16 program priorities from 15 (high) to 0 (low). An idle loop runs at priority -1 when the system has nothing else to do. These priorities pertain to core resident and mass memory resident programs. When a program is running at its priority, it can be suspended by any higher priority program which the system allows to run. The suspended program waits in core to resume execution when the priority structure works back down to it.

The priority structure can only be changed by interrupts (hardware) or scheduling (software). The mask in the M register allows hardware, which has a higher priority than the running program, to interrupt. When an interrupt occurs, the Common Interrupt Handler saves all the registers of the interrupted program on the Interrupt Stack (so that the program can later be resumed) and transfers control to the program which will service the interrupt on the line.

A schedule request allows a program to change the priority level. If the request is for a higher level program, a pseudo interrupt will occur immediately (the suspended program goes on the Interrupt Stack) and the higher level program is executed. Control will later return to the suspended program. If the schedule request is for an equal or lower level priority, the request parameters go on the Scheduler Stack and are threaded in it by priority to be picked up later when the priority structure works down to that level.

All programs exit to the MSOS Dispatcher. It is the Dispatcher that must cause the next lowest priority program to be executed. It does this by looking at the Interrupt Stack and Scheduler Stack and finding the highest priority waiting program.



11.1 SOURCE, OBJECT AND ABSOLUTIZED PROGRAMS

11.1.1 Source Program

The source program is the program written in assembly language code by the programmer. It most likely would be punched on cards or on paper tape. Here is an example of a source program listing:

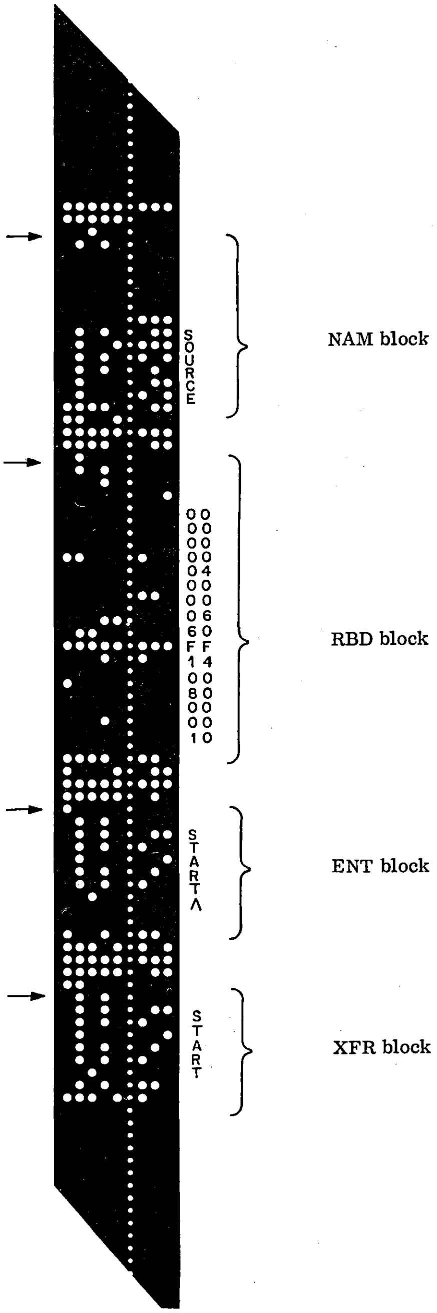
```
0001                          NAM    SOURCE
0002                          ENT    START
0003 P0000 0000     START     0      0
0004 P0001 C400               LDA*   X
     P0002 0006 P
0005 P0003 60FF               STA-   I
0006 P0004 1400               JMP*   (START)
     P0005 8000 P
0007 P0006 0010     X         NUM    $10
0008                          END    START


I       00FF   START    0000P X         0006P
```

The source program is read into the computer by the assembler. The computer does not execute the program at this time. The assembler translates the mnemonics into binary object code. The source program is listed at this time (on the teletypewriter or the line printer) and the object program is punched on paper tape (or on the disk, drum or magnetic tape).

11.1.2 Object Program

It is important to know that the object program is not executed either. It must be loaded by a "relocating" loader back into the computer before it can be executed. It is not important at this point to be able to interpret the codes in the object program. One must just understand that the codes represent the desired program and that the loader will interpret the codes when it loads the object program and will make an executable program out of these codes. On the following page is an example of how the previous source program would look in object form.

Figure 24. Object Tape



Note that each block on the paper tape is preceded by the one's complement of the word count in that block and followed by a checksum word on that block.

Figure 24. Object Tape (Cont)

1 word = 2 frames on paper tape

NAM BLOCK

0010	0000	0101	0000
0	0	0	0
0	0	0	0
0	0	0	7
S		O	
U		R	
C		E	

RBD BLOCK

0100	0000	0101	0000
0001	0000	0000	0001
0	0	0	0
0	0	0	0
C	4	0	0
0	0	0	6
0000	0000	0001	1000
6	0	F	F
1	4	0	0
8	0	0	0
0	0	1	0
0	0	0	0

ENT BLOCK

1000	0000	0101	0000
S		T	
A		R	
T			
0	0	0	0

XFR BLOCK

1100	0000	0101	
S		T	
A		R	
T			

The following is some of the information about the blocks the assembler makes in the binary object program for the loader. This was extracted from the Loader Chapter of the MSOS Reference Manual, and it may be consulted for more detail.

Relocatable Binary Input

The loader recognizes relocatable binary blocks by the type indicator field in bits 13-15 of the first word of the block. The following block types are defined:

Type	Indicator	Description
NAM	001	Name block
RBD	010	Command sequence block
BZS	011	Zero storage block
ENT	100	Entry point block
EXT	101	External name block
XFR	110	Transfer address block

If the loader is unable to recognize the indicator, it does not process the block.

NAM Block

The NAM block contains a word count for common storage and data storage, the program length, and the name of the program.

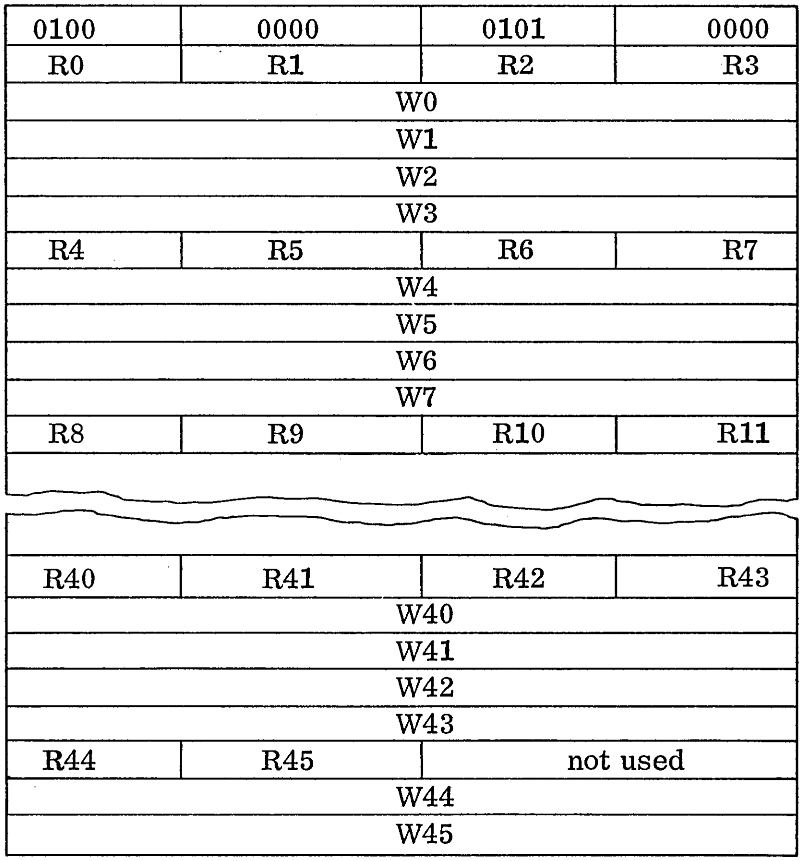
<table>
<tr><td>0010</td><td>0000</td><td>0101</td><td>0000</td><td></td></tr>
<tr><td colspan="4">Number of words in common storage block</td><td></td></tr>
<tr><td colspan="4">Number of words in data storage block</td><td></td></tr>
<tr><td colspan="4">program length</td><td></td></tr>
<tr><td colspan="2">character 1</td><td colspan="2">character 2</td><td rowspan="3">Program Name</td></tr>
<tr><td colspan="2">character 3</td><td colspan="2">character 4</td></tr>
<tr><td colspan="2">character 5</td><td colspan="2">character 6</td></tr>
</table>

RBD Block

An RBD block contains a portion of the actual command sequence data of the program. Words 2-59 contain the relocation bytes and words for the command sequence input. Each relocation byte is a 4-bit indicator that identifies a word of the command sequence input as an absolute 15-bit address or as a 15-bit address relative to some relocation base. The relocation base for a word is determined by the particular combination of bit settings within the relocation byte.

Relocation bytes in RBD blocks:

0000	Absolute (no relocation)
0001	Positive program relocation
0101	Negative program relocation
0010	Positive common storage relocation
0110	Negative common storage relocation
0011	Positive data storage relocation
0111	Negative data storage relocation



0100	0000	0101	0000
R0	R1	R2	R3
W0			
W1			
W2			
W3			
R4	R5	R6	R7
W4			
W5			
W6			
W7			
R8	R9	R10	R11
R40	R41	R42	R43
W40			
W41			
W42			
W43			
R44	R45	not used	
W44			
W45			

Core Image of RBD Block

Wn	nth word of input block, (n=1-45)
Rn	Relocation byte of nth word
W0	Origin address of input block
R0	Relocation byte for W0

There is one relocation byte for every word in the command sequence input, and a maximum of 45 words in an RBD block. The first word is the address at which the loader begins storing command sequence data. The relocation byte for the first word address (storage address) of an RBD block may be 0000, 0001, or 0011. Zero is the leading bit for all but the last relocation byte; one is the leading bit for the last relocation byte.

In processing an RBD block, the loader picks up the 15 bits which represent the first word address of the command sequence data in the block. It adjusts this address for relocation according to the setting of the bits representing its relocation byte. The resulting absolute address is the first word address in core to receive the command sequence data (stored in consecutive locations). Each word is relocated according to its relocation byte.

ENT Block

1000	0000	0101	0000
name 1	character 1	character 2	
	3	4	
	5	6	
E1			
name 2			
E2			

name 13	
E13	
name 14	
E14	
not used	
not used	

Core Image of ENT Block

Namen = Six-character name of nth entry in block
En = Entry point address of nth name

XFR Block

The XFR block contains a transfer address (in words 2-4), which is six ASCII characters in length, including trailing spaces. The transfer address must be an entry point in the program being loaded or in another program loaded during the same load operation.

1100	0000	0101	0000
Character 1		Character 2	
	3		4
	5		6

Core Image of XFR Block

The XFR block must be the last in a relocatable binary program. If an XFR block is out of order, a loader error message is issued and the load is terminated. The loader records the transfer address in the XFR block. If two or more relocatable binary programs are loaded with one operation, the loader saves the last transfer address for the start of execution.

It is obvious by looking at the code that it could not be executed exactly as it appears. Normally the programmer never has to know what the object tape looks like. He would only have to know the format if he wanted to examine parts of the tape. For example, if he stored a number of object programs on one tape, he may wish to be able to search the tape for a particular program. He could do that by writing a program to look at the NAM blocks until it found the right one.

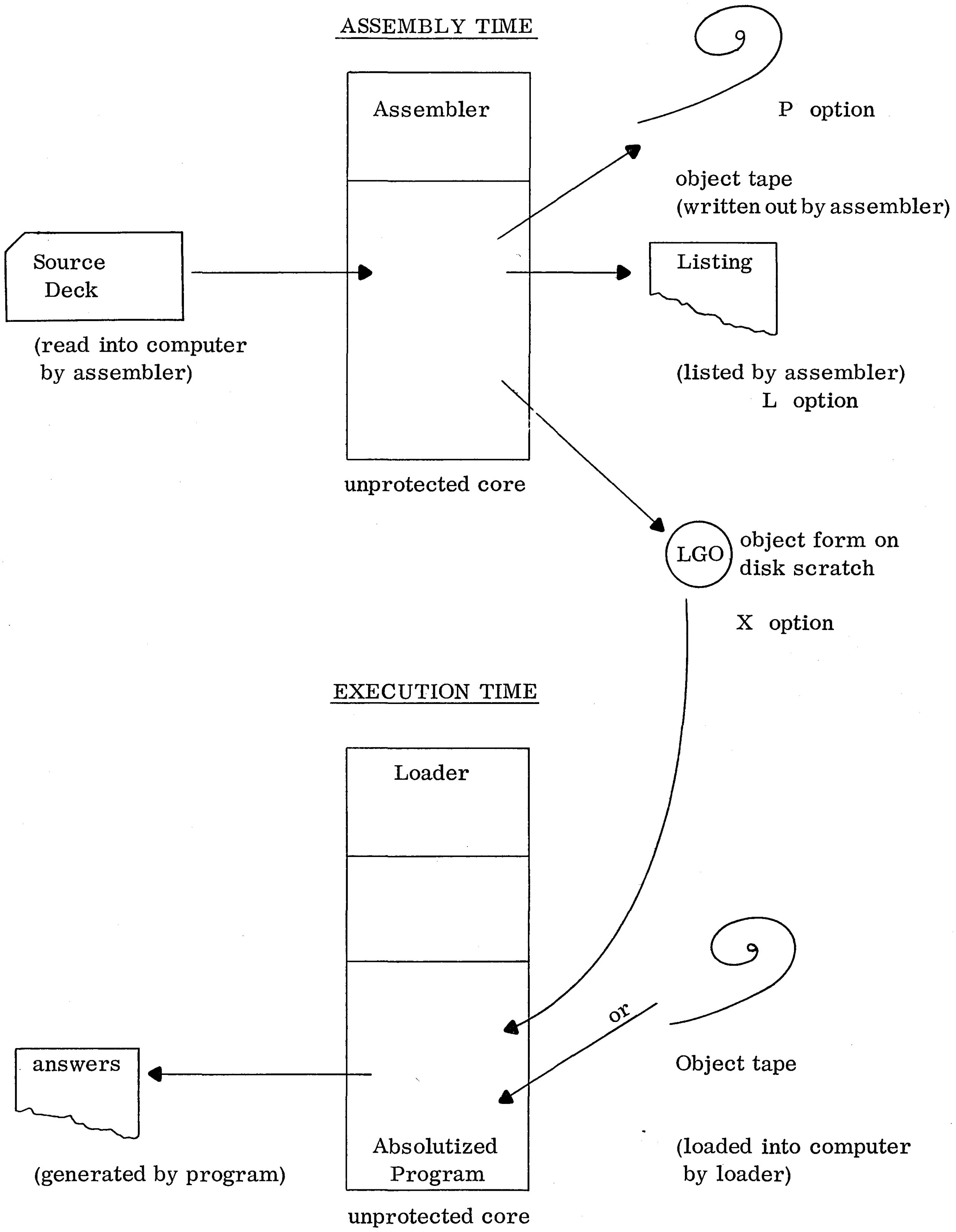
11.1.3 Absolutized Program

The programmer is, however, very much concerned with how the program looks after it is loaded into core for execution. The different addressing modes he used when he wrote the source program will determine what the final core image of that program looks like. An absolutized program is an exact copy of this core image, which can be executed. The loader loads and absolutizes the object program at a certain address, which is wherever it happens to load it. It also links the program to any externals and loads and links any library subroutines required. A utility routine could be used to punch a tape with this core image on it; hence, the term "absolute tape". Here is an example of an absolutized image of the previous object program, (if the program was loaded at $1000).

```
($1000)     0000          0       0          Op Code
($1001)     C400          LDA+    X
($1002)     1006
($1003)     60FF          STA-    I
($1004)     1400          JMP+    (START)
($1005)     9000
($1006)     0010          NUM    $10
```

Notice that the contents of location $1002 is $1006 to indicate where X is and that $1005 contains $9000 to indicate the jump through location $1000. The 0006P and 8000P in the source code were relocated by the loader when the object program was loaded. No matter where the program was loaded, the correct addresses would be filled in at that time.

Figure 25. Flow of Program Through Execution



11.1.4 Form of Programs in MSOS Libraries

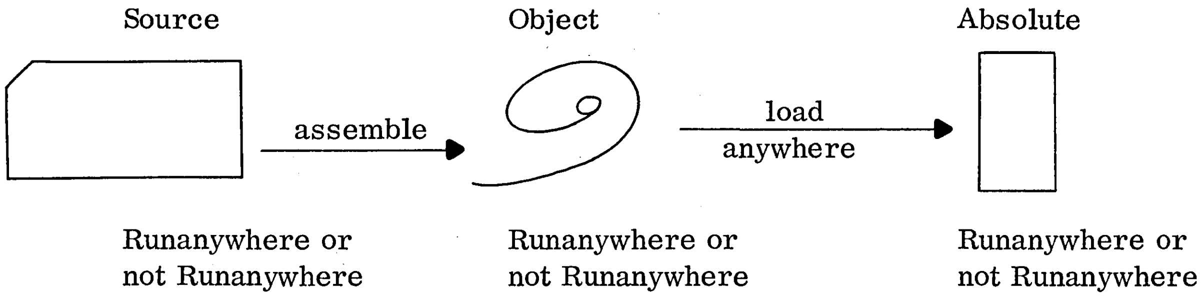
The reason it is so important for the analyst to understand the distinction between what the object program generated by the assembler and the absolutized program after loading look like is because user and system programs are stored in the libraries on mass storage in these two different forms. User background programs in the program library are stored in relocatable binary object form. They will be loaded by the loader into core whenever they are executed, so they will be absolutized and linked each time they are loaded and run.

System programs (including user process programs) in the system library are stored in absolutized form and will be read into core (without any changes) whenever they are needed for execution. This is because it must be possible to bring the real time process programs into memory very fast; the relative time it would take to load them in with the loader every time they are needed would be too great. A much better solution would be to write the source program in such a fashion that the absolutized program could be run anywhere in core and would still execute properly. The absolutizing of the system process programs is done during system initialization (by the loader portion of the system initializer) when the programs are stored on the system library.

11.2 RUN ANYWHERE CODING

11.2.1 Writing Programs for Run Anywhere Coding

The method devised for writing programs so they can be stored in absolute form and still run anywhere in core is called Run Anywhere Coding. It is important to know that this is done at the source level.



The object program can be loaded anywhere and absolutized and will run correctly at that time because the loader has relocated any addresses which were in the program. However, if an absolute image of this program is later run somewhere else in core, it will run correctly if it was coded run anywhere in the original source form. Here is an example of the same program coded both ways; (assume it was loaded and absolutized at $1000).

Not Runanywhere

```
     Source                                      Absolute

START     0         0                     ($1000) = 0 0 0 0
          LDA+      X                     ($1001) = C 4 0 0
                                          ($1002) = 1 0 0 6
          STA-      I                     ($1003) = 6 0 F F
          JMP+      (START)               ($1004) = 1 4 0 0
                                          ($1005) = 9 0 0 0
X         NUM       $10                   ($1006) = 0 0 1 0
```

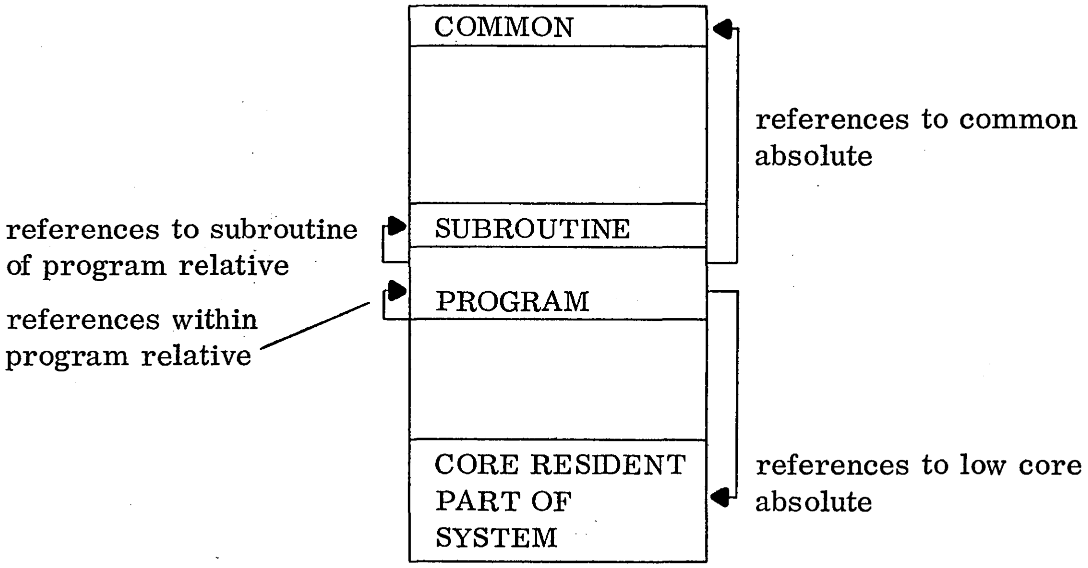
Runanywhere

```
     Source                                      Absolute

START     0         0                     ($1000) = 0 0 0 0
          LDA*      X                     ($1001) = C 8 0 3
          STA-      I                     ($1002) = 6 0 F F
          JMP*      (START)               ($1003) = 1 C F C
X         NUM       $10                   ($1004) = 0 0 1 0
```

Notice that the addresses of X and START in the first example were relocated by the loader to show that X is at $1006 and START at $1000. The program will run at $1000, but if, for example, it is moved to $2000 without the object being reloaded, it will not run correctly because it will think X is at $1006 (when actually it moved to $2006) and then it will jump through $1000 (when actually the entry point START moved to $2000).

However, in the second example all addressing in the program is relative. The LDA* loads from X which is 3 locations forward and the JMP* jumps through START which is 3 locations backward. Yet the STA- I must be left as it was because the I register is absolute core location $FF, and it will always be there. The program is runanywhere because it can be kept in absolute form and can be later run anywhere in core with correct results.

One might at this point wonder why not code all programs in run anywhere form. The primary reason is that it is more difficult to learn to do run anywhere coding since there are more chances for the programmer to make errors which will not produce any error messages. In general, when writing a program to be run anywhere, all references to addresses that move with the program should be relative, and all addresses which are absolute core locations must be absolute.


COMMON
references to common absolute
references to subroutine of program relative
SUBROUTINE
PROGRAM
references within program relative
CORE RESIDENT PART OF SYSTEM
references to low core absolute

```
            NAM        RA
            ENT        SCAN
            EXT*       ALARM          relative external
            EXT        LOWPGM         absolute external
            COM        X(10)
SCAN        0          0
            LDA+       X+0            reference common abso-
                                      lute
            STA-       $FF            reference $FF absolute
            LDA+       $10C           reference interrupt trap
                                      absolute
            RTJ        ALARM          reference subroutine
                                      relative
            RTJ+       LOWPGM         reference program in
                                      system resident absolute
            END
```

Without worrying about the externals at this point, note that all references within the program area must be relative. A good way to tell if there are any which are not relative is to look at the source listing and see if any of the codes on the left are followed by a P, i.e., in the program SOURCE example, change:

```
      P0001  C400   LDA+        X
      P0002  0006P
to
      P0001  C803   LDA*        X
```

Change all loads, stores, jumps, etc., in the program to relative.

Look at the VALUE problem from chapter 6 and observe the addressing used in the program. COUNT is addressed in two-word relative mode, because the DATA block most likely moves with the program and may be further away than 127_{10} locations. LPMASK+6 is addressed with one-word absolute mode because it is a fixed low core address. MASK in the program is addressed in one-word relative mode because it moves with the program. Yet X in the common block must be addressed in two-word absolute mode because common is fixed and is in high core. The LDA VALUE is in two-word relative mode. This implies VALUE is relative to the TEST subroutine and the assembler requires two-word addressing for any relative externals.

11.2.1

VALUE PROBLEM

```
                NAM         TEST
                COM         DUMMY(10),X(10)
                DAT         DUM(6),COUNT(1)
                EQU         LPMASK($2)
                ENT         START
                EXT*        VALUE
MASK            BZS         MASK(1)
START           0           0
                CLR         Q
                STQ         COUNT
                LDA         VALUE
                AND-        LPMASK+6
                ALS         8
                STA*        MASK
                ENQ         9
SEARCH          LDA+        X,Q
                AND         =N$3F00
                EOR*        MASK
                SAN         2
                RAO         COUNT
                SQZ         EXIT-*-1
                INQ         -1
                JMP*        SEARCH
EXIT            JMP*        (START)
                END
```

11.2.2 Buffer Addresses

Buffer addresses used for indirect addressing (to load or store) in the program must be absolutized each time they are used. If the buffer is in low core system resident, it can be absolutely addressed:

```
           |
           |
          EXT            BUF
BUFADR    ADC            BUF
           |
           |
           |
          STA*           (BUFADR)
           |
           |
```

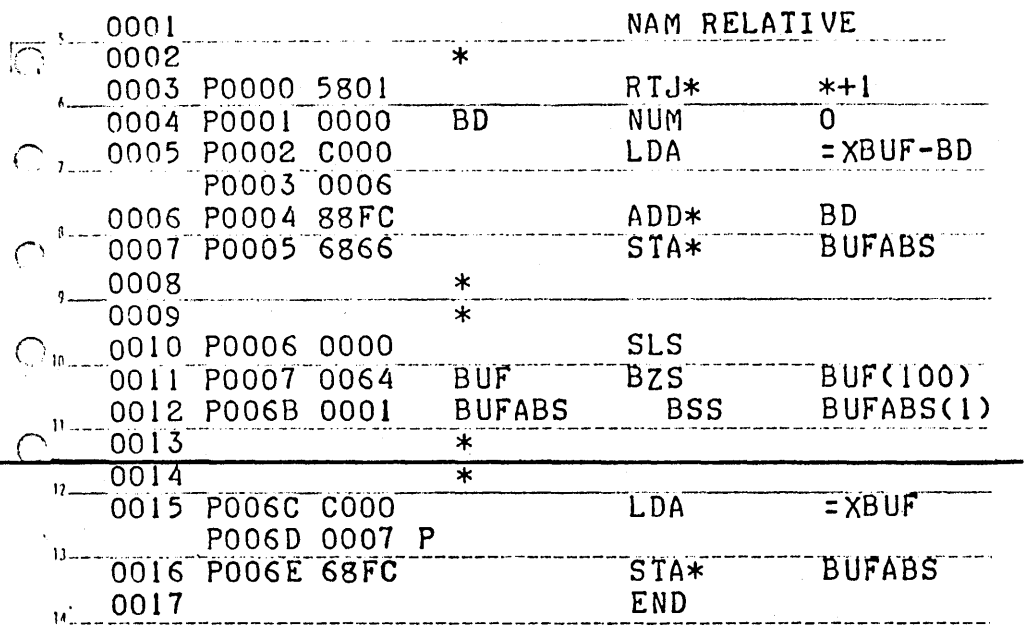
The address assembled into BUFADR will be the absolute address of the buffer. Since the buffer will never move, even though the program moves, the addressing will still be correct.

However, if the buffer moves with the program, the ADC would not work because it would contain the address of where the buffer was when the program was absolutized. In the following example RELATIVE the buffer BUF is in the program at P0007. BUFABS must always contain the address of where the buffer really is. If we used:

```
BUFABS    ADC            BUF
```

BUFABS would assemble as 0007P which, if the program was absolutized at $1000, would contain $1007. Whenever the program moved, the buffer would not be at $1007 any more. Neither would the example at P006C work correctly for the same reason.

Study the code beginning at P0000 and see that BUFABS is calculated each time it is used. The RTJ* will cause the current address of the BD instruction tobe stored in BD. (RTJ stores P+1 in jump address, then RNI at P+2.) Then the LDA calculates the distance from BUF to BD. ADD the contents of BD (the current address of BD) and store it in BUFABS to calculate the buffer address!



```
CORRECT        0001                                    NAM RELATIVE
               0002                        *
               0003 P0000 5801                         RTJ*         *+1
               0004 P0001 0000      BD                 NUM          0
               0005 P0002 C000                         LDA          =XBUF-BD
                    P0003 0006
               0006 P0004 88FC                         ADD*         BD
               0007 P0005 6866                         STA*         BUFABS
               0008                        *
               0009                        *
               0010 P0006 0000                         SLS
               0011 P0007 0064      BUF                BZS          BUF(100)
               0012 P006B 0001      BUFABS               BSS        BUFABS(1)
               0013                        *
INCORRECT      0014                        *
               0015 P006C C000                         LDA          =XBUF
                    P006D 0007 P
               0016 P006E 68FC                         STA*         BUFABS
               0017                                    END
```

The buffer address will be correct whether the buffer is forward from the LDA =XBUF-BD or behind it, because of the 15-bit arithmetic used. With BUF in its current location the calculation would be

```
 0007
-0001
 0006   assembled
+0001
 0007
```

If the buffer were at 7FFE relatively, the calculation would be

```
 7FFE
-0001
 7FFD   assembled
+0001
 7FFE
```

A very nice way to clean up even the correct code above would be:

```
               RTJ*           BUFABS
BUF            BZS            BUF(100)
BUFABS         0              0
```

The RTJ jumps over the buffer, taking the buffer address with it. It stores it in BUFABS, then continues execution following BUFABS.

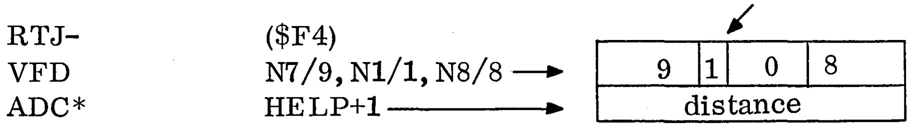
Look at the peripheral program BOOTSTRAP for the paper tape reader in chapter 8. Location P0012 is used to contain the buffer address of where the data is to read into. Since this address is program relocatable (contains 0014P), the program is not run anywhere because when it is loaded and absolutized P0012 will contain the absolute buffer address at that time. If the program and buffer were moved, ADDRES would then contain the wrong buffer address.

More coverage of the externals for run anywhere programs will be mentioned in a later section, 11.6. How the externals are written depends primarily on where they and the referencing program are in the system.

11.2.3 X Bit in System Requests

In run anywhere programs, the X bit must be used in system requests; and addresses used in the requests must follow the same rules for addresses in the program which move relative to the program or are external in low core.

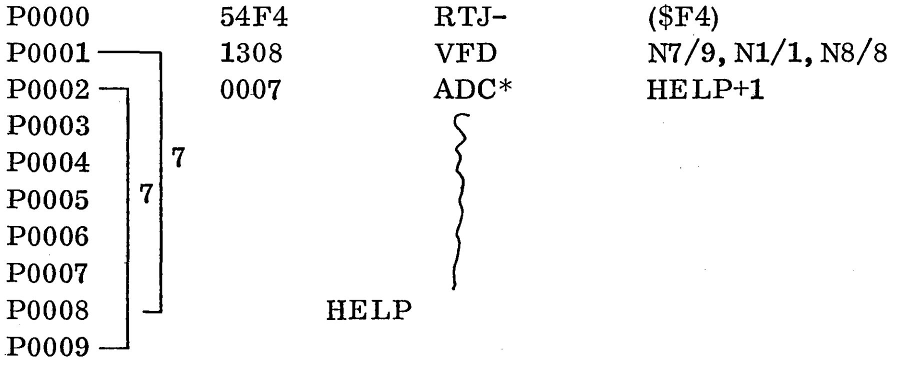
The following example is for the X bit being set in a schedule request:



```
RTJ-        ($F4)
VFD         N7/9,N1/1,N8/8 →
ADC*        HELP+1 ————————→
```

Since this is a request made from a run anywhere program, the address being scheduled, which is HELP, must be relative if HELP is relative to this request. (HELP could be in the scheduling program or external to it.)

When the X bit is used as above to indicate relative addresses, the relative distance must be from the first word of the parameter string (not from the RTJ or the ADC). That is why the relative address constant (ADC* HELP+1) is coded as HELP+1 instead of HELP. The distance established from the ADC to HELP+1 would be the same distance as from the VFD to HELP, which is the distance required by the operating system.



```
P0000       54F4        RTJ-        ($F4)
P0001       1308        VFD         N7/9,N1/1,N8/8
P0002       0007        ADC*        HELP+1
P0003
P0004
P0005
P0006
P0007
P0008            HELP
P0009
```

Of course, if HELP were a core resident program, it would have to be scheduled absolutely from a run anywhere program. It can be scheduled this way even if it is in the system directory.

```
          54F4      RTJ-        ($F4)
          1208      NUM         $1208
          7FFFX     ADC         HELP
                    {
                    EXT         HELP
```

When the X bit is set to indicate relative, all addresses in that request must follow suit. In an I/O request, for example, the completion address and buffer address must be relative. Again, they must be relative to the first word of the parameter string.

```
          RTJ-      ($F4)
REQ       NUM       $0D56       FWRITE,RP=5,CP=6
          ADC*      COMPL+1
          NUM       0
          NUM       $18FC       ASCII, STD PRINT DEV
          NUM       10
          ADC*      BUF+5
          {
COMPL
          {
BUF
```

Note that COMPL+1 is used to add 1 to the distance between the ADC* and COMPL (to make it relative to REQ). BUF+5 is used to add 5 to the distance between the ADC* and BUF. The number of words does not have to be relative even though the X bit is set.

If it is desired to use relative addressing to locate the number of words, it would be done as follows:

```
          RTJ-      ($F4)
REQ       NUM       $0D56
          ADC*      COMPL+1
          NUM       0
          NUM       $18FC
          ADC*      (N+4)
          ADC*      BUF+5
COMPL     {
BUF
N         NUM       10
```

All other requests which can utilize the X bit for relative addressing must follow the same pattern.

The ADC* in the above examples will only calculate the correct distance for addresses forward from the ADC*. Since the relative address must be a 15-bit positive increment which is added to P and must work whether the distance is forward or backward, the following method is often used:

```
                    ADC          COMPL-*+1
or
                    ADC          BUF-*+5
```

The regular ADC form used here has an expression in the address field which will always calculate the correct distance. The desired address minus the current P counter makes a 15-bit relative distance from P and the adjustment of +1 or +5 in the example is for a completion or buffer address.

Figure 26 shows incorrect examples of relative addressing. See MMPGM in the mass memory coding section 11.5.7 for more examples. Relative addressing is detailed in Chapter 8 of this manual. Also, the MSOS Reference Manual contains details of using the X bit in all system requests.

Figure 26. Error Examples for Incorrect Addressing in Mass Memory Programs

NOTE: Buffer is at P0018
Completion address is at P0012

```
0013                          FWRITE    $FC,WROTE,MSGBUF,10,A,5,6,I,1
0013  P0002  54F4
0013  P0003  0D56  <--------  X bit set
0013  P0004  0012 P )
      P0005  0000   |
0013  P0006  18FC   }         But address still program relocatable
0013  P0007  000A   |
      P0008  0018 P ) <-----  Buffer was at P0018 in early example

0013                          FWRITE    $FC,WROTE-*+1,(MSGBUF),10,A,5,6,I,X
0013  P0002  54F4
0013  P0003  0D56  <--------  X bit set
0013  P0004  000F  <--------  Rel. dist. to WROTE
      P0005  0000
0013  P0006  18FC
0013  P0007  000A
      P0008  8018 P <-------  Indirect bit on buffer address
```

NOTE: Buffer is at P0003
Completion address is at P001D

```
0016                    WRITE    FWRITE   $FC,*-WROTE-5,*+MSGBUF-5,10,A,5,6,I,X
0016  P000D  54F4
0016  P000E  0D56
0016  P000F  7FEC  <------------  Wrong rel. dist. to compl.
      P0010  0000
0016  P0011  18FC
0016  P0012  000A
********RL*********
      P0013  0000  <------------  Illegal relocation to buffer

0016                    WRITE    FWRITE   $FC,*-WROTE-5,*-MSGBUF-5,10,A,5,6,I,X
0016  P000D  54F4
0016  P000E  0D56
0016  P000F  7FEC  <------------  Wrong rel. dist. to compl. (appears backward)
      P0010  0000
0016  P0011  18FC
0016  P0012  000A
      P0013  000B  <------------  Backward rel. dist. to buffer (appears forward)
```

Problem: Write a runanywhere program.

Given: skeleton of a program which computes an average of 10 positive numbers.

```
            NAM         AVERAGE
            ENT         AVG
            BZS         OVFL(1)
AVG         0           0
            ENQ         9
            ENA         0
            SOV         0
LOOP        ADD*        X,Q
            SNO         TEST-*-1
            RAO*        OVFL
            AND         =N$7FFF
TEST        SQZ         AV-*-1
            INQ         -1
            JMP*        LOOP
AV          LDQ*        OVFL
            ALS         1
            LRS         1
            DVI         =N10
            JMP*        (AVG)
            END         AVG
```

11.2.3

a. Write a main program, with a buffer with data in it, to call AVG as a subroutine. Set up the proper linkage between the main program and its subroutine. The main program should punch the answer (the average of the data) on binary paper tape.* Be sure the programs work before going further.

b. The programs should be coded in runanywhere form and should not destroy themselves.

c. To check out the runanywhere features of the programs add a move subroutine to move the main program and AVG to a higher core area after they have run once and given one answer. Then control should be transferred to the entry point of the main program at its new address to run it again and see if it gives the same answer.

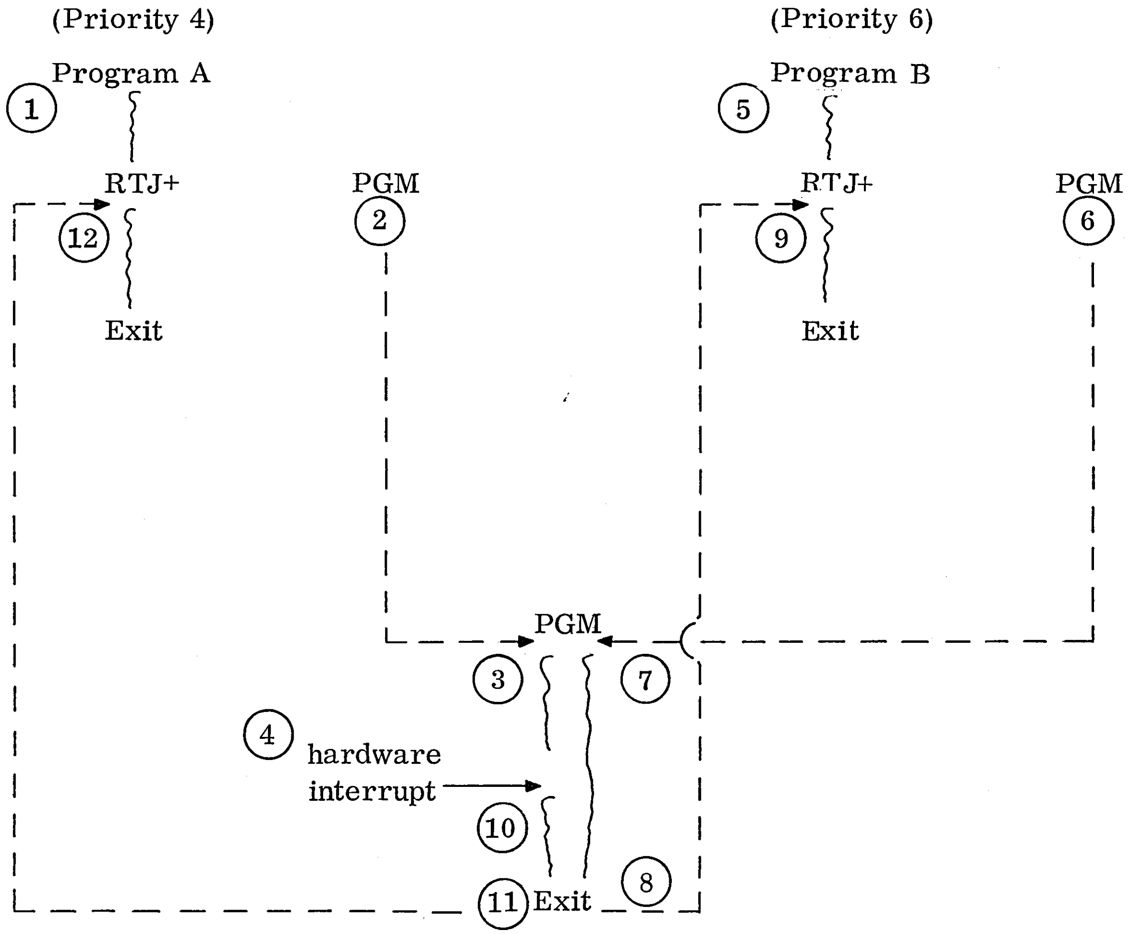
This will simulate a runanywhere mass memory module being executed in a different core area, and it can be checked out in the background with the protect switch set.

If CONVRT is used, it should not be moved in the move and should be addressed absolutely. This is because it is not runanywhere. Using CONVRT would simulate a mass memory module calling a core resident subroutine which remains at a fixed location even though the module runs in different locations.

11.3 REENTRANT CODING

It is necessary in a real time process environment for many of the programs to be reentrant. This is because the process programs run at different priority levels and may have common subroutines. A reentrant program is one that can be entered at more than one priority level. The program may begin its computations but be stopped (perhaps as a result of a hardware interrupt at a higher level). Then it may be entered again at a higher level (perhaps by being called by the higher level interrupting program). It must do a computation for the higher level calling program. Then it must resume the original computation later without losing any continuity or results. An example of a situation in which a subroutine PGM must be reentrant is as shown on the following page.

*The main program could instead call the CONVRT conversion subroutine to convert the hexadecimal answer to ASCII codes, then write it in ASCII on the teletype.



1. Program A runs at Priority 4 and
2. calls PGM (at same priority).
3. PGM is running when
4. an interrupt occurs.
5. Program B begins to run at priority 6 (higher) as a result of the interrupt, and
6. it, too, calls PGM.
7. PGM must run a calculation for program B and
8. return to program B.
9. Program B must complete and exit. At that time
10. the priority drops back down to 4, and PGM resumes its computation for program A where the interrupt occurred. It must correctly complete its run for A and
11. return to program A. A then can
12. complete and exit. PGM must be reentrant so it can make correct computations and exits for A and B.

11.3.1 Methods of Reentrants

There are a number of different methods which are used to make programs reentrant. 1700 MSOS provides for reentrant programs by containing a core area called Volatile Storage which any protected program may use. Volatile storage is actually a BSS block in the program VOLA and its size is set up at system initialization time. A program can establish its reentrancy by requesting a temporary area of volatile storage for each run in which to store its temporary results during execution. No locations in the program area can be used for temporary results because they would be destroyed if the program was reentered before it completed execution. Therefore, all data would be either in volatile storage or in the registers. (P,A,Q and I would be saved in the interrupt stack if an interrupt occurred.)

The following program can be used as an example and it will be recoded to be reentrant. The addresses of two parameters which are to be added together by the subroutine are passed in A and Q. The answer is to be passed back in A.

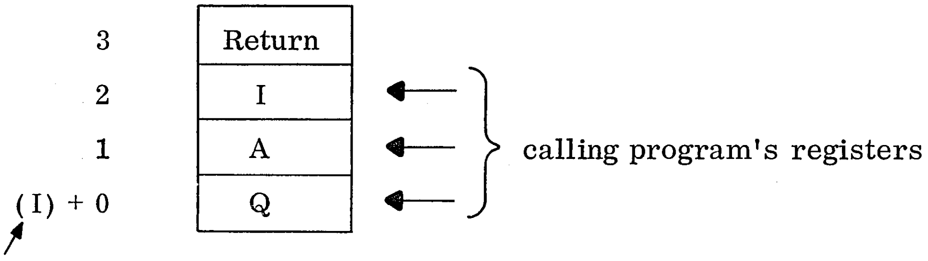
```
            NAM       ADD2
            ENT       ADD2
ADD2        0         0
            STA*      TEMP          STORE ADDRESSES OF
            STQ*      TEMP+1        PARAMETERS.
            LDA*      (TEMP)        PICK UP PARAM 1
            ADD*      (TEMP+1)      ADD PARAM 2
            JMP*      (ADD2)        RETURN WITH ANSWER
             ⋮                      IN A
            BSS       TEMP(2)
            END
```

The program as it is written above would not be reentrant because its return address and parameter addresses would be lost if the program was reentered before it was finished.

The following is the program coded in reentrant form.

```
                 NAM       ADD2
                 ENT       ADD2
                 EQU       AVOLA($BB),AVOLR($BA),ZERO($22)
        ADD2     0         0          ENTRY POINT
                 IIN                  LOCK OUT INTERRUPTS
                 RTJ-      (AVOLA)    GO GET SOME VOLATILE
                 NUM       4          4 WORDS WANTED                ENTRY
                 EIN
                 LDA*      ADD2       PICK UP RETURN ADDRESS
                 STA-      3,I        SAVE IT IN VOLATILE
                 LDA-      (ZERO),Q   GET PARAM 2 IN A
                 LDQ-      1,I        GET PARAM 1 ADDRESS IN Q
                 ADD-      (ZERO),Q   ADD PARAM 1 TO PARAM 2
                 STA-      1,I        PASS ANS. BACK IN A
        EXIT     LDQ-      3,I        PICK UP RETURN ADDRESS
                 IIN
                 STQ*      ADD2       STORE RETURN ADDRESS
                 RTJ-      (AVOLR)    GIVE BACK VOLATILE            EXIT
                 EIN                  ENABLE INTERRUPTS
                 JMP*      (ADD2)     EXIT
                 END
```

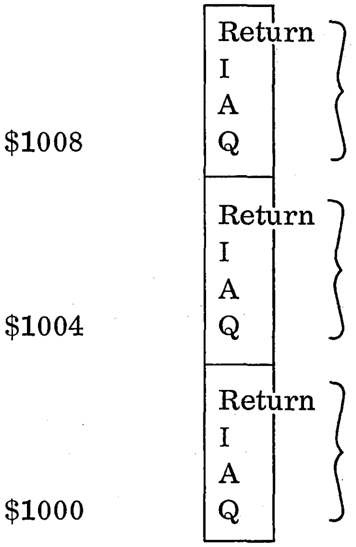
4-word volatile storage block



Subroutine ADD2's I register contains volatile address.

A separate 4-word block of volatile storage will be assigned each time the program is reentered. Since its I register is always saved when an interrupt occurs, the value of I in each run will locate the specific block being used in that run.

Successive blocks allocated will always be to higher priorities and, naturally, release will be in reverse order.



Volatile assigned on 3rd entry; i.e., at priority 6, I register contains $1008

Volatile assigned on 2nd entry; i.e., at priority 5, I register contains $1004

Volatile assigned on 1st entry; i.e., at priority 4, I register contains $1000

Lines 1 through 7 in the example ADD2 program could be the same in any reentrant program consisting of getting volatile storage for that run and saving the return address. Lines 12 through 17 could be the same also, consisting of returning the volatile and exiting. Lines 8 through 11 comprise the program itself. Note that in this case no TEMP's were needed because the parameter addresses in the calling program's A and Q registers are contained in volatile locations 1 and 0 and they can be used to access the parameters. The following is a line-by-line description of the reentrant coding for the ADD2 program.

1. The entry point is entered by a RTJ from the calling program.

2. The subroutine must lock out interrupts because the return address which the RTJ stored in the entry point must not be lost. Since the RTJ instruction stores its return address in the jump address and does an RNI at jump address+1 (line 2, the IIN) an interrupt cannot occur and wipe out the return address in ADD2 before the IIN. The IIN must also be in effect before the RTJ to VOLA because VOLA is not reentrant and expects interrupts to be locked out before it is entered.

3,4 Jump to VOLA (through locore location $BB). Pass a parameter to VOLA requesting four words of volatile storage. A different block will be assigned each time this subroutine is entered, and the address of the block comes to the calling program (ADD2) in I. ADD2 must never destroy its I register because it contains the address of the volatile block. VOLA also saves the calling program's Q, A and I registers in the first three words of the block.

All the registers in ADD2 (including I) will always be safe from an interrupting and reentering program because ADD2's registers would be saved on the interrupt stack.

5. Enable interrupts as soon as possible because it is illegal to lock them out for more than 50 µs (including the time in VOLA).

6. One free instruction is allowed after the EIN before an interrupt can occur; so the return address is rescued from the entry point -- it will be safe in A. At any time after this point an interrupt can occur and the Interrupt Handler will save the registers.

7. Store the return address in volatile so that the A register can be used in the program.

8. Q still has the address of parameter 2 in it from the calling program so it can be used to get the parameter into A.

9. The address of parameter 1 was passed in A and VOLA put it in the 2nd word. It can be loaded into Q.

10. Parameter 1 is then added into A using the address in Q. Note in 8 and 10 that the addresses are put in Q. One way of addressing the parameters which would <u>not</u> work is:

```
LDA-        (I+1)
ADD-        (I+0)
```

It looks like it would work because the addresses <u>are</u> in I+1 and I+0 but this would be assembled as $100 and $FF which of course is <u>not</u> where the addresses are. An assembly error would probably occur at the I+1. Another method which would not work is:

```
LDA-        ($3), I
ADD-        ($22), I
```

Location $3 has a 1 in it to which will be added the <u>contents</u> of $FF (giving I+1) and the add would be from 0+ the contents of $FF (giving I+0). This would get the parameter <u>address</u> in A. Then the ADD would add in the second parameter <u>address.</u>

11. The answer is stored in volatile+1 because that is where VOLR will restore the A register from. Any parameters to be returned in registers must be put in the first three words of volatile.

12. The return address must be picked up out of volatile because volatile is going to be returned and it would be lost.

13. Interrupts are locked out for exiting.

14. The return address is stored in ADD2 because the original Q is going to replace ADD2's Q.

15. Volatile is returned. VOLR restores the original registers from the first three words of volatile! That is why the answer was put there, to get it in the original A. Also, since volatile is now gone, the return address had to be rescued; and it could not be left in Q (to exit through Q) because ADD2's Q is gone.

16. Enable interrupts.

17. Free instruction to exit.

<u>Incorrect</u> example:

```
line 10          ADD-       (ZERO),Q      ANSWER IN A.
                 LDQ-       3,I           RETURN ADDR IN Q
                 RTJ-       (AVOLR)       RETURN VOLATILE
                 EIN
┌──────────────► JMP*       (ZERO),Q
│
```

Q now has the original Q in it, not the return address; also the answer in A was clobbered and has the original A in it.

If any other temporary locations were needed by the reentrant program, they would be requested in volatile and would be addressed the same way. For example, if it was desired to move the parameter addresses to volatile +5 and 6 (like TEMP's in non-reentrant program), the NUM in line 4 would be 6 and lines 8 through 12 would be replaced with:

```
LDA-      1,I          PICK UP PARAM 1 ADDRESS
STA-      4,I          MOVE TO TEMP (I+4)
LDQ+      0,I          PICK UP PARAM 2 ADDRESS
STQ-      5,I          MOVE TO TEMP+1(I+5)
LDQ-      4,I          GET PARAM 1 ADDRESS IN Q
LDA-      (ZERO),Q     GET PARAM 1
LDQ-      5,I          GET PARAM 2 ADDRESS IN Q
ADD-      (ZERO),Q     ADD PARAM 2 TO PARAM 1
STA-      1,I          STORE ANSWER TO PASS IN A
```

5	address of parameter 2
4	address of parameter 1
3	return
2	I
1	A
(I) + 0	Q

This example coding (immediately above) is used simply to illustrate how the program can access volatile for its temporary results. Of course, it would be inefficient to code the present example this way because the move and reloading are not necessary. Study all the addressing carefully.

11.3.2 Reentrant Problem, AVG

Following is the AVG program which computes an average. Rewrite the program as a non-destructive and reentrant subroutine. Assume that the calling program passes the number of words in A and the first word address in Q to the subroutine. The subroutine should pass the average back in A and the remainder in Q. Remember that interrupts may not be inhibited more than 50 μs at any one time.

```
             NAM        AVERAGE
             ENT        AVG
             BZS        OVFL(1)
AVG          0          0
             ENQ        9
             ENA        0
             SOV        0
LOOP         ADD*       X,Q
             SNO        TEST-*-1
             RAO*       OVFL
             AND        =N$7FFF
TEST         SQZ        AV-*-1
             INQ        -1
             JMP*       LOOP
AV           LDQ*       OVFL
             ALS        1
             LRS        1
             DVI        =N10
             JMP*       (AVG)
             END        AVG
```

11.4 MSOS REQUESTS MADE BY SYSTEM PROGRAMS

Remember that all requests to MSOS simply put a desired action (such as a Write or Schedule) on a list (queue) and that the action may or may not be performed (depending on the rules) before control returns to the requestor.

There are special considerations system programs (either core resident or mass memory resident) must make when making requests to the operating system. This is in addition to the considerations and rules jobs must follow.

First, all protected programs must check a request after it is made to see if it was accepted. When control is returned to the requestor beneath the parameter string, Q will be positive if the request was accepted, negative if the request was rejected. For example:

```
                  {
                  RTJ-          ($F4)     )
                  NUM           $1206     }    schedule request
                  ADC           PGM       )
        CKQ       SQP           REQOK-*-1
                  JMP*          REQREJ
        REQOK     {
```

In the above example when control is returned at CKQ, a skip is made to REQOK if, indeed, the request was accepted. Otherwise, (Q negative) a jump is made to REQREJ because the request is rejected. If a request is rejected, the program might either repeat the request at REQREJ, change the priority, or print an error message. Remember that when a job made a request, the system would continually repeat the request until it was accepted, then return control to the program beneath the parameter string. However, all protected programs should check every request in the above fashion since there will be no other indication if a request is rejected.

Even though a bit of Q is used to indicate acceptance or rejection of a request by the system, if the program were passing a parameter in Q, the full 16-bit original Q is still passed intact. Only the upper bit has been changed when control comes back to the requestor.

11.4.1 Schedule

A schedule request made by a system program would be coded the same as one made by a job except the requestor should check to see that it is accepted. Examples would be:

```
                  {
                  RTJ-          ($F4)
                  NUM           $1205
                  ADC           WRITE
                  SQP           OK-*-1
                  JMP*          REJ
        OK        {
```

WRITE is in the scheduling program. It will be executed at priority 5.

```
                {
                EXT           MIPRO
                {
                RTJ-          ($F4)
                NUM           $1204
                ADC           MIPRO
                SQP           OK-*-1
                JMP*          REJ
       OK       {
```

MIPRO is external to the scheduling program and will be run at priority 4. It is core resident and not in the system directory.

```
                {
                EXT           MIPRO
                {
                RTJ-          ($F4)
                NUM           $1204
                ADC           (MIPRO)
                SQP           OK-*-1
                JMP*          REJ
       OK       {
```

Here MIPRO is in the system directory and will run at 4. The MIPRO program could be either core resident or mass memory resident. The scheduling program does not have to know where MIPRO is as the system will find it.

There is an important consideration to make when deciding whether to schedule a program with the regular external form or whether to schedule it with the system directory form (having the program in the system directory). If it is undecided whether the scheduled program will be core resident or mass memory resident, use the system directory form as it will work in either case. The same logic would apply to a core resident program which may later be a mass memory program. Using the system directory form simply causes the system to have four more words of core (for the directory entry) and take a few microseconds longer to schedule but this can save miles of recoding.

11.4.1.1 Priorities (Schedule)

The system program will be concerned with the priorities of the scheduled programs. The software priorities run from 15 (highest) to 0 (lowest). The CP (completion priority) field in the request is the desired priority for the address scheduled. This means that scheduling is actually like jumping to an address and changing the running priority concurrently.

If the priority in the schedule request is lower than or equal to the current running priority of the program making the request, the scheduled program will run after the current program is finished and the priority works down to it. It waits by priority in the scheduler stack with other scheduled programs. This is the way to set up things to be done after the current program exits.

If the scheduled priority is higher than the running priority, a pseudo interrupt occurs immediately and the scheduled program is executed. Then control returns back to the requestor.

Process programs usually run at priorities of 4, 5 and 6. Jobs run at 0 and 1. Other priorities are usually for the operating system and hardware interrupts.

11.4.1.2 Rejects (Schedule)

A scheduled request for a core resident system program which is not in the system directory would be rejected if there is no room in the scheduler stack (the queue for waiting scheduled programs) for the request parameters. If the request is for a program which is in the system directory and that program is already threaded on some thread, a reject would occur. This implies that the program has been scheduled (perhaps by some other program) and not yet finished. Parameters for schedule requests for system directory programs are not transferred to the scheduler stack so a reject would not occur as a result of the scheduler stack being full.

At this point it could be noted that relatively minor modifications to the system would be required to allow for a queue of requestors waiting to rerun a desired system directory program (for example, a mass memory program which is already in core). It would even be possible to allow that mass memory program to be reentered by a higher priority interrupting program (in which case it would have to be reentrant).

11.4.2 TIMER Request

Since TIMER requests are simply delayed scheduler calls, they may be made by system programs with the same considerations as required for scheduler calls.

11.4.3 I/O Requests

The I/O requests (READ, WRITE, FREAD, FWRITE) must consider priorities when these requests are in protected programs。 They must also check to see if the requests were accepted. The actual operation of what the requests do is the same as for jobs.

The following is an FWRITE example:

```
            RTJ-        ($F4)
            NUM         $0C76        FWRITE, RP=7, CP=6
            ADC         WROTE        COMPLETION
            NUM         0
            NUM         $18FC        ASCII MODE, STD. PRINT DEVICE
            NUM         35           35 WORDS
            ADC         BUF          BUFFER ADDRESS
            SQP         OK-*-1
            JMP*        REJ
OK          ⸾           ⸾

WROTE       ⸾           ⸾
BUF         BSS         BUF(35)
            END
```

11.4.3.1 Priorities (I/O)

The request priority (RP=7 in example) has absolutely nothing to do with the running priority of the program. It is the priority of this request with respect to other requests for a logical unit number. In other words if there are RP=6 and RP=10 requests waiting to be written on the printer, the RP=7 requests will be threaded in between the 10 and the 6. Each logical unit has a queue of requests waiting for it and the driver will handle them sequentially by priority. When the write actually gets done is a function of the driver (program) priority and its relationship with the priority of the running program. For example, if the teletype driver runs at priority 10 and the program runs at priority 6, the driver will periodically interrupt the program to do the actual write operation.

The completion priority in the requests (CP=6 in example) is related to the running priority of the program. It is like a schedule request for the completion address after the I/O is finished. In the example, if the running priority is 4, a pseudo interrupt will occur and the priority will be changed to 6 when the I/O is finished and the completion routine entered.

11.4.3.1

From the time the request is initiated until the time the I/O is completed, the thread word in the request will be non-zero. Here is an example of a program which runs at priority 4, initiates a request, and loops waiting for the request to be finished.

```
         RTJ-      ($F4)
         NUM       $0CE0        FWRITE, RP=14
         NUM       0            NO COMPLETION
THREAD   NUM       0            THREAD
         NUM       $0804        ASCII, OUT ON TTY
         NUM       35
         ADC       BUF
         SQP       LOOP-*-1     REQUEST ACCEPTED
         JMP*      REQREJ
LOOP     LDA*      THREAD       THREAD ZERO YET?
         SAZ       COMPL-*-1    SKIP OUT; I/O DONE
         JMP*      *-2
COMPL
```

Many programs have been coded this way; and this should not be done. Looping like this at any priority is going to slow down a system by locking out lower priorities. For example, if many process programs were coded this way, they could almost completely lock out job processing (which runs at 0 and 1). It would be much better to code the write with a completion address and jump to the dispatcher to wait for the completion routine to be entered.

It is important to control the priorities in a program which makes a number of I/O requests. Completion routines should be very short and should be at a higher priority than the rest of the program to cause a software interrupt in the program when I/O is complete. Completion priority could be lower than running priority if it is simply desired to check for errors at the end of the program or if it is desired for the programs handling the data to run lower.

If a program is to run at priority 4 and all its I/O completion is to run at priority 5, it would be necessary for each completion routine to schedule the priority back down to the program priority before initiating the next request.

```
            EXT         REJ, IOERR
            EQU         ADISP($EA)
             ∫
            RTJ-        ($F4)
            NUM         $0875        FREAD, CP=5
            ADC         COMPRD       COMPLETION
            NUM         0
            NUM         $1005        ASCII,  LUN 5=CR
            NUM         40           40-WORD BUFFER
            ADC         BUF
            SQP         REQOK
            JMP+        REJ
REQOK       JMP-        (ADISP)

COMPRD      SQP         SCHPRT       PRIORITY HERE IS 5
            RTJ         IOERR
SCHPRT      RTJ         ($F4)
            NUM         $1204        SCHEDULE, DOWN TO 4
            ADC         PRINT        PRINT IN SAME PGM
            JMP-        (ADISP)
PRINT       RTJ-        ($F4)        PRINT NOW AT PRIORITY 4
            NUM         $0C05        FWRITE, CP=5
            ADC         COMPPR
            NUM         0,$1004,35   LUN 4 (TTY), 35 WORDS
            ADC         BUF
             ∫
```

11.4.3.2 Rejects (I/O)

An I/O request could be rejected if the request is already threaded (like if the program tried to start up that write again before it was finished) or if the system tries to schedule the driver and finds the scheduler stack full. (In that case the driver's priority would be lower than the running priority of the program and that is not normal.)

11.4.3.3 System Request Problem, THREAD

The following example program runs at priority 12 and makes a Write request for the teletypewriter (logical unit 4) at priority 14. Since it has no completion address, it does not jump to the dispatcher to wait for the Write to be finished as control would not return to the program. Instead, it waits for the Write to be finished by looping on the thread word at LOOP. (The thread word was filled when the request was made and becomes zero again only when the driver has finished this request.) The driver runs at priority 10.

```
          RTJ-        ($F4)
          NUM         $0CE0          FWRITE, RP=14
          NUM         0              NO COMPLETION
THREAD    NUM         0              THREAD
          NUM         $1004          ASCII, OUT ON TTY
          NUM         35
          ADC         BUF
          SQP         LOOP-*-1
          JMP*        REQREJ
LOOP      LDA*        THREAD
          SAZ         COMPL-*-1
          JMP*        *-2
COMPL
```

When will the write actually be done?

11.4.4 EXIT Requests

All programs exit to the dispatcher. This is so the dispatcher can pick up the highest priority program waiting to be executed next, whether it is a previously interrupted program or a scheduled program. This is how the priority drops.

The EXIT request made by an unprotected program generates a jump to the dispatcher. A protected program may not use the EXIT request; it must code the jump to the dispatcher:

```
          {
          EQU         ADISP($EA)
          {
          JMP-        (ADISP)
          END
```

11.4.5 SPACE and RELEASE

There are two requests which only protected programs are allowed to make: Space and Release. These requests are used to get, and later release, core in the protected area called allocatable core. Any unprotected program may access this area and may use the space for anything it desires -- to contain data or programs. The allocatable core area is divided into priority blocks (so that some core will always be available at the highest priorities). The sizes of the blocks at each priority are set up by the systems analysts at system initialization time.

11.4.5.1 SPACE Request

Here is the format of the SPACE request:

RTJ- ($F4)

15	9	8	4	0
	RC=10		RP	CP
COMPL				
THREAD = 0				
Q				
N WORDS				

The Macro form for the SPACE request would be:

SPACE n, compl, rp, cp, X

RC is request code 10. Bit 8 is X bit.

RP is priority of the block in which space is desired.

CP is the priority of the completion address.

COMPL is the completion address of where control will go after the space has been gotten.

THREAD is the thread word, zero.

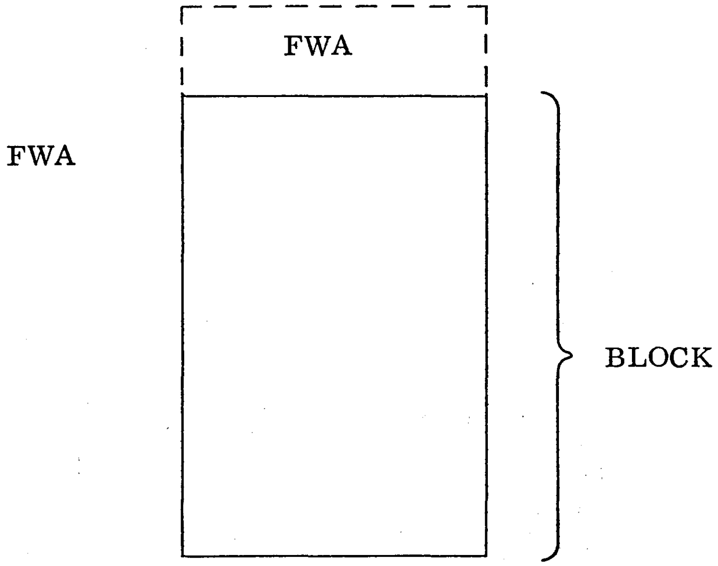
Q is the Q register parameter passed back to the completion routine. Q will contain:

address - of the space gotten, or
$8000 - if no space will ever be available at this priority; this means that even if the background were swapped out there would still not be enough core to fill the request.

N WORDS is the number of words of space requested.

The address of the space would also be in core immediately preceding the space block which was acquired.

11.4.5.1



A swap would occur if necessary to get core under the following conditions:

Request priority is greater than 3
Completion priority is greater than 2
Core is not already swapped
No unprotected I/O is going on
The minimum time between swaps is passed

Otherwise, the request for space waits on a queue.

The space request would only be rejected if it was already threaded. (i.e., still involved in a previous operation.)

11.4.5.2 RELEASE Request

After the program which requested the space is finished using it, it must release the space. This is done with a RELEASE request. The format of the release request is as follows:

RTJ- ($F4)

15	9	8	1	0
	RC = 12		0	t
FWA				

The Macro call for RELEASE is:

RELEAS fwa,t,x

RC is request code 12; bit 8 is X bit.

't' bit, bit 0, should be 0 if it is desired for control to return to the program (releasor) after the release is made. The 't' bit is set to 1 if control should go to the dispatcher instead. The release request is the only one which allows this choice of whether to come back to the program beneath the parameter string (as all other requests do) or to go to the dispatcher if this was the last thing to do. The 't' bit will be used in coding mass memory programs.

FWA is the address of the core to be released. It must be the correct address of the block or else no release will occur and there will be no error message. This is to provide flexibility so that a program which was coded to run on mass memory (in a later section) could be changed to be core resident without any changes or reassembly of the program. When it became core resident and tried to release its core, the core simply wouldn't be released.

The release request causes the space to be immediately given to any other space requestor waiting on the space queue before control returns to the releasor (or dispatcher).

An example of a space and release request could be:

```
                   RTJ-     ($F4)
1.                 NUM      $1445        RP=4, CP=5
                   ADC      COMPL        COMPLETION ADDRESS
                   NUM      0
                   NUM      0            ADDRESS COMES BACK HERE
                   NUM      $1000        $1000 WORDS WANTED
2.                 SQM      REJ          REQUEST REJECTED
                   JMP-     (ADISP)      WAIT FOR SPACE
3.   COMPL         SQP      GOTSP-*-1    GOT SOME SPACE
                   JMP*     NOSPAC       NO SPACE GOTTEN
4.   GOTSP         STQ*     REL+2        ADDR. OF SPACE IN RELEASE
5.



6.   REL           RTJ-     ($F4)        GO RELEASE SPACE
7.                 NUM      $1800
8.                 NUM      0            ADDRESS OF SPACE
9.





                   END
```

1. Space request, priority 4 block, completion priority 5
2. Q is checked to see if request was not accepted.
3. Completion address is entered when space is gotten or if there is no space. Q would be negative if no space was obtained.
4. If space was obtained, Q contains the address of the block. Here it is stored in the release request. The address could also have been obtained later from word 3 of the space parameter list.
5. Continue in program, using space.
6. Release the space; program is finished using it.
7. 't' bit is not set, so control returns to the program after the release.
8. The address of the block is here; it was placed here after the space was obtained.
9. Continue in program.

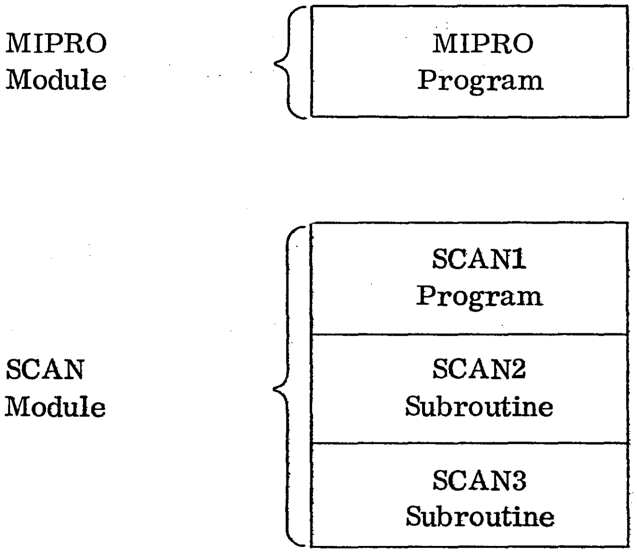
Note that the above program is not runanywhere so it must be a core resident system program. The completion address in the space request is not relative.

11.5 CODING MASS MEMORY PROGRAMS

All programs that are to be part of the System Library resident on mass memory must conform to special rules. All of the rules are logical when the inter-relationship of the program and the system is considered. The most important general consideration is to be sure the program gets to do everything it set out to do before it disappears.

11.5.1 Modules in Library

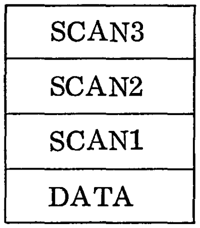
The programs are stored on mass memory in absolutized form. (The System Initializer put them there.) Each program - or a set of a program and its subroutines together - is called a module and has a name unique to the module. The name must not appear as an entry point anywhere in the system. The name of the module must be in the system directory. Here is an example of two modules on mass memory:



(MIPRO is not an entry point.)

MIPRO is the program which handles manual interrupts to the process; the module is made up of the single program. SCAN is a module made up of a user process program SCAN1 and its subroutines SCAN2 and SCAN3. These three programs go together and will always be together as a group when the module is brought into core for execution.

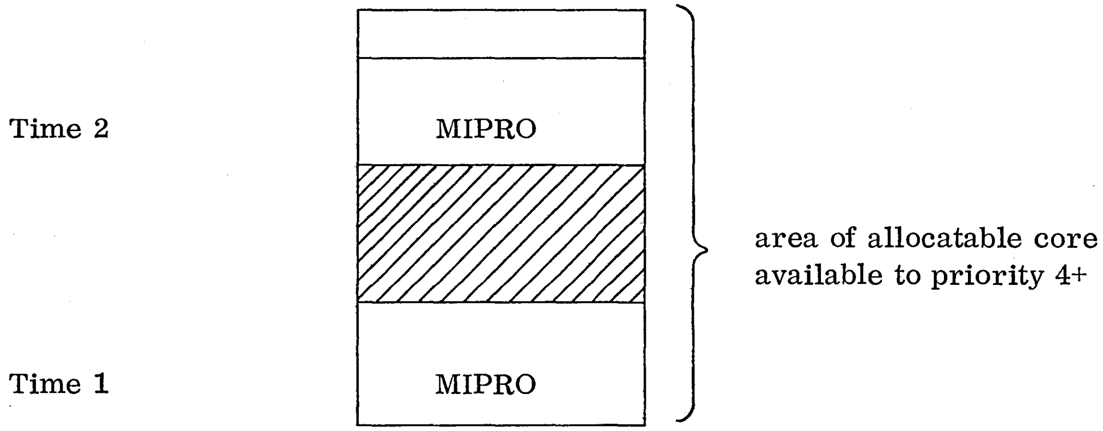
It is possible for a mass memory module to contain a DATA block within it, accessible only to the programs in that module. Any number of modules may contain separate DATA blocks, but there may be only one in each module. Here is an example of the SCAN module if it contained a DATA block:



The programs in any module may use system COMMON, which is in highest core.

11.5.2 Allocatable Core

The area of core that the mass memory programs run in is called allocatable core. It is divided into priority blocks and the highest core area is available to the largest priority programs. A program to be run will be put in the smallest space it will fit in which is available to that priority. Note that this means a program may run at different places in allocatable core at different times. There is no dynamic relocation of the programs in allocatable core so as core spaces are released, they are saved for subsequent programs to be run in.



In the above example MIPRO may run in different places depending on the space available.

The systems analysts decide what priority area the program will run in (at system initialization time); so a program calling a mass memory program has no control over this.

11.5.3 Scheduling the Mass Memory Program

When it is desired to bring a mass memory program into core for execution, the calling program schedules it in. An example would be:

```
EXT             MIPRO
 ⋮
SCHDLE          (MIPRO),4,0
```

The system program MIPRO is scheduled in, to be executed at priority 4. As in all requests for system programs, MIPRO must be named external and must be in parentheses in the SCHDLE request.

The name MIPRO is the name of the module in the system directory. This request causes the system to obtain space in allocatable core to put the program in (by a SPACE request), then to read it in (by a mass memory READ request), then transfer control to it at its first core location.

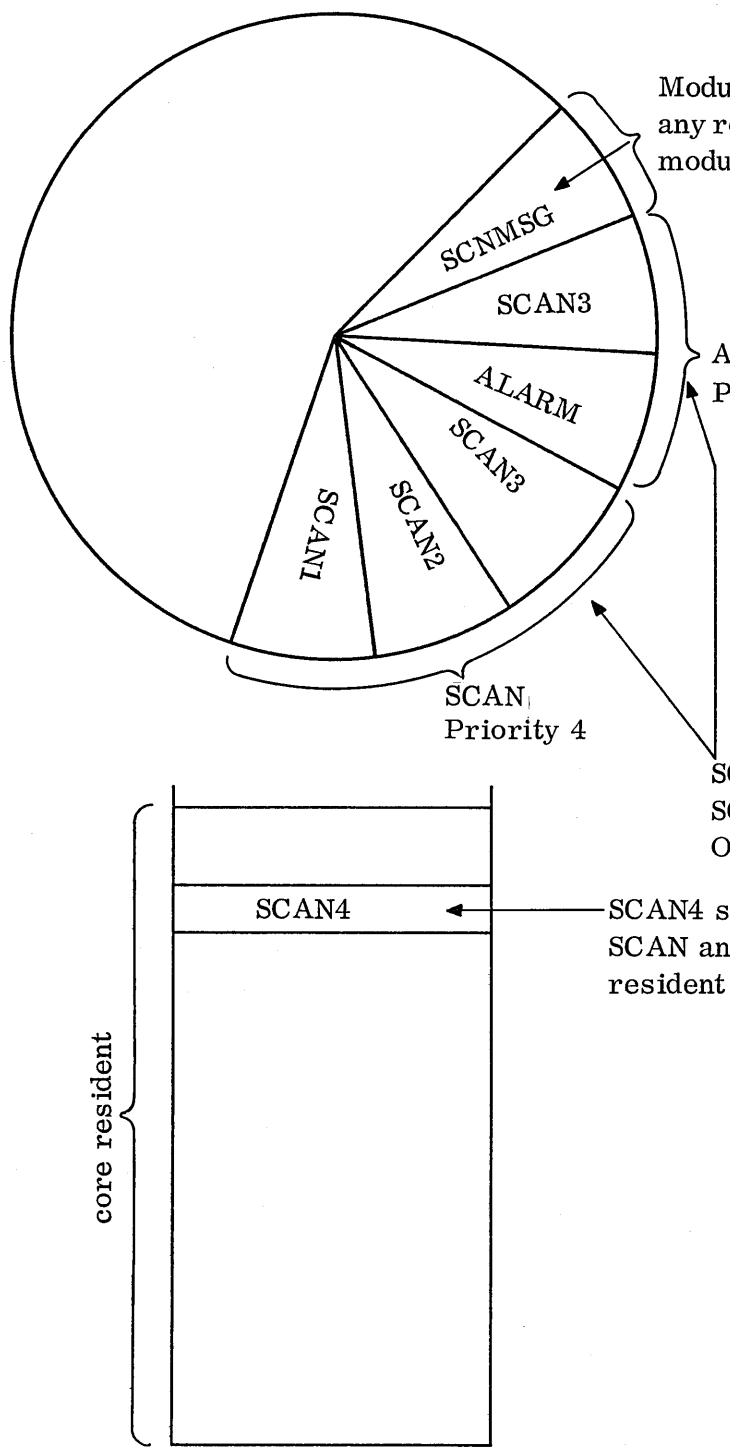
The calling program does not have to worry about all this; it simply knows that scheduling it causes the program to come in to core and begin execution. The schedule request would be rejected if the program has already been scheduled (by another caller) and is still in the process of being brought into core. It could not be rejected from the scheduler stack being full because the parameters are not transferred to the stack.

11.5.4 Form of Mass Memory Programs

All mass memory programs have to be runanywhere, as has already been covered. This is because they are stored on the system library in absolute form and are run at different places in allocatable core, not where they were absolutized.

Mass memory programs do not normally have to be reentrant because a program is usually brought into core each time it is called to be run. Minor modifications to the system would be required to allow a mass memory program which is in core to be reentered by a higher priority interrupting program; in that case it would have to be reentrant.

Figure 27. Maps of Mass Memory Modules and Core Subroutines



Module SCNMSG could be scheduled by any routine in either SCAN or ALARMS module.

SCAN3 subroutine needed by both SCAN and ALARMS modules -- One copy in each module.

SCAN4 subroutine also needed by both SCAN and ALARMS modules. It is core resident and reentrant.

11.5.5 Externals to Mass Memory Programs

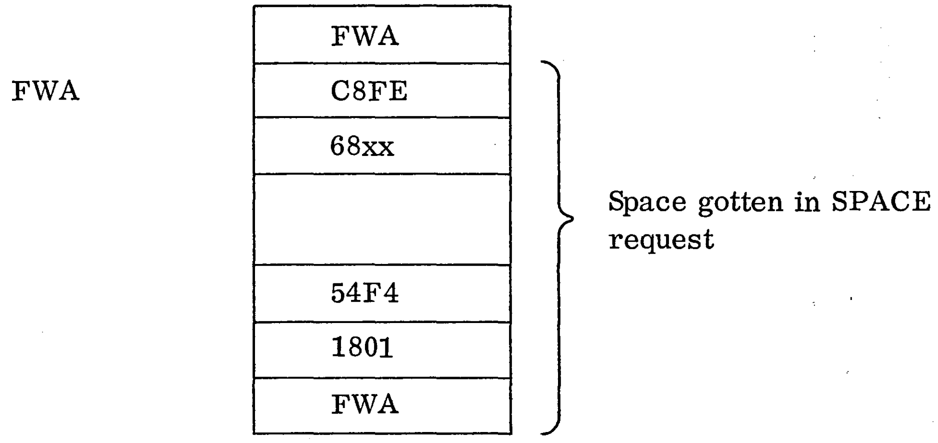
Externals in mass memory programs which reference locations which are core resident must be absolute. Externals which reference addresses in other mass memory programs (in the same module) must be relative. There must not be any externals which reference addresses in any other module. This is because one module does not know when another module is in core (or where it is) unless special links are provided to handle it. This means that any subroutine which several modules need would be either core resident or there would be a separate copy in each module that needs it. Another solution would be to put the subroutine in a separate module by itself and let routines needing it schedule it.

11.5.6 Space

Mass memory programs may release their own space. This is a good feature because it means that a program can schedule a mass memory program and then can forget about it after the schedule request has been accepted and exit knowing that the scheduled program will be executed at its priority. The mass memory program could pick up the address of where it is, as its first instruction (i.e.), and store it in the release request which would be the last instruction in the program.

```
1.                NAM       SCNMSG
2.   SCNMSG       NUM       $C8FE       LDA* *-1
                  STA*      REL+2
                   {
     REL          RTJ-      ($F4)
3.                NUM       $1801       RELEASE REQ., TBIT SET
                  NUM       0           ADDRESS TO BE RELEASED
                  END
```

core during execution:



1. Note that SCNMSG is not declared as an entry point since it is also the name of the module.

2. Remember that the first word preceding Space in allocatable core obtained by a SPACE request contains the address of where that block of space is. The NUM $C8FE is to fake out the assembler and cause it to make a code as the first word of the program which will be a LDA* *-1. A LDA* *-1 instruction would not have been assembled properly because it attempts to reference outside of the program area relatively. Also remember that the first word of a mass memory module is executed so the NUM will not be treated as data. So, the $C8FE gets the address of the space occupied by the program into A.

3. Note that in the release request the 't' bit (bit 0 in the first word of the parameter string) is set to indicate to the system not to return to the program which made the release request after releasing the space. This bit would be necessary for a mass memory program releasing its own core. It makes sense because if control was returned to the program, there isn't any program after the end. Or, even if there was some more program (such as a jump to the dispatcher), it may not be executed. Remember that when a release request is made, the space is allocated to any waiting requests before the return to the requestor. Therefore, the space may have been given away and may in fact contain another program or data; a return to the releasing program would cause a mess because it may not be there any more.

   If the mass memory module contains several subroutines, the NUM $C8FE would be the first instruction in the module and the RELEASE would be the last request made in the module.

   Mass memory programs should complete their I/O and their calls to any subroutines before exiting. This is because when they release their space and exit, any data buffers or completion addresses in the program may be lost as soon as the release request is made.

11.5.7 Mass Memory Problem, MMPGM

The following example program contains one error which could cause incorrect results. It is very subtle and difficult to locate.

Assume that the assembly-language coding is runanywhere and correct. Look for an error which can occur during execution. The program runs at priority 4. The driver runs at priority 10. The completion routine runs at priority 6.

11.5.7

```
                        NAM     MMPGM             MASS MEM PGM EXAMPLE
                        ENT     MMPGM
                        EXT*    REQREJ,IOERR      MM EXTS IN SAME MODULE RELATIVE
                        EXT     CORSUB            CORE RES SUB EXT ABSOLUTE
                        EXT     SYSPGM            SYS DIR PGMS MUST BE EXTERNAL
               *
      00FA     ADISP    EQU     ADISP($FA)
               *
               *                                  FIRST INSTR OF MM PGM EXECUTABLE
P0000 C8FE     MMPGM    NUM     $C8FE             PICK UP CORE ADDRESS OF MMPGM
                        STA*    REL+2
                        JMP*    WRITE
               *
P0003 4D41     MSGBUF   ALF     *,MASS MEMORY EXAMPLE*
P0004 5353
P0005 204D
P0006 454D
P0007 4F52
P0008 5920
P0009 4558
P000A 414D
P000B 504C
P000C 4520
               *
               WRITE    FWRITE    $FC,WROTE-*+1,MSGBUF-*+5,10,A,5,6,I,X
P000D 54F4
P000E 0D56
P000F 000F
P0010 0000
P0011 18FC
P0012 000A
P0013 7FF4
P0014 0162              SQP       REQOK           PROT PGMS MUST CHECK REQ ACC
P0015 5800  X           RTJ       REQREJ          GO HANDLE REJ;REL ADDRESSING
P0016 7FFF  X
P0017 5400  X  REQOK    RTJ+      CORSUB          ABS ADDRESSING TO CORE RES PGM
P0018 7FFF  X
P0019 14FA              JMP-      (ADISP)         NO MORE TO DO
                        SCHDLE    (SYSPGM),4,0    SCH SYS DIR PGM, CORE OR MM
P001A 54F4
P001B 1204
P001C FFFF  X
               *
P001D 0162     WROTE    SQP       REL             CHECK Q FOR IO ERRORS
P001E 5800  X           RTJ       IOERR           GO ANALYZE ERRORS
P001F 7FFF
               *
               REL      RELEAS    0,T,0           0 WILL BE REPLACED W/ADDRESS
P0020 54F4
P0021 1801
P0022 0000
               *                                  DO NOT RETURN TO PGM AFTER REL
                        END       MMPGM
```

Map

```
I          00FF    MMPGM    0000P  ADISP   00FA   MSGBUF   0003P  WRITE    000DP
REQOK      0017P   WROTE    001DP  REL     0020P  SYSPGM   001CX  CORSUB   0018X
IOERR      001FX   REQREJ   0016X
```

The following is a description of the concepts presented in MMPGM. Note that an entry point MMPGM is declared; therefore, the name of the module must be something other than MMPGM. There are two relative externals, REQREJ and IOERR, which must be subroutines in this same module. There is one absolute external, CORSUB, which must be a core resident subroutine.

P0000 - picks up the address of the core block MMPGM is located in, then stores it in the release request.

P000D - initiates an FWRITE from MSGBUF. RP=5 and CP=6. The X bit is set so WROTE-*+1 and MSGBUF-*+5 calculate the correct distances from P0Q0E to the completion address WROTE and the buffer MSGBUF and places them in P000F and P0013.

```
 000E                         000E
+000F                        +7FF4
 ----                         ----
 001D   WROTE                 8002
                                 1
                              ----
                              0003   MSGBUF
```

P0014 - checks to see if the request was accepted.

P0017 - jumps to execute CORSUB at running priority 4.

P0019 - schedules a program in the system directory, SYSPGM, to be executed at priority 4 after completion of MMPGM. Note that SYSPGM must be named external and that the ( ) causes bit 15 to be set in P001B. The program should have checked Q to see if the request was accepted.

P001C - the program has now run out of things to do, so it goes to the dispatcher to await the write completion.

P001D - after the write, the completion routine (running at priority 6) checks for errors.

P0020 - the completion routine then releases core. The T bit is set so that after core is released, control will go to the dispatcher instead of returning to the program.

11.6 EXTERNAL REFERENCES AND LINKAGE, SUMMARY

It is desirable at this point to review the rules regarding externals among the various types of system programs.

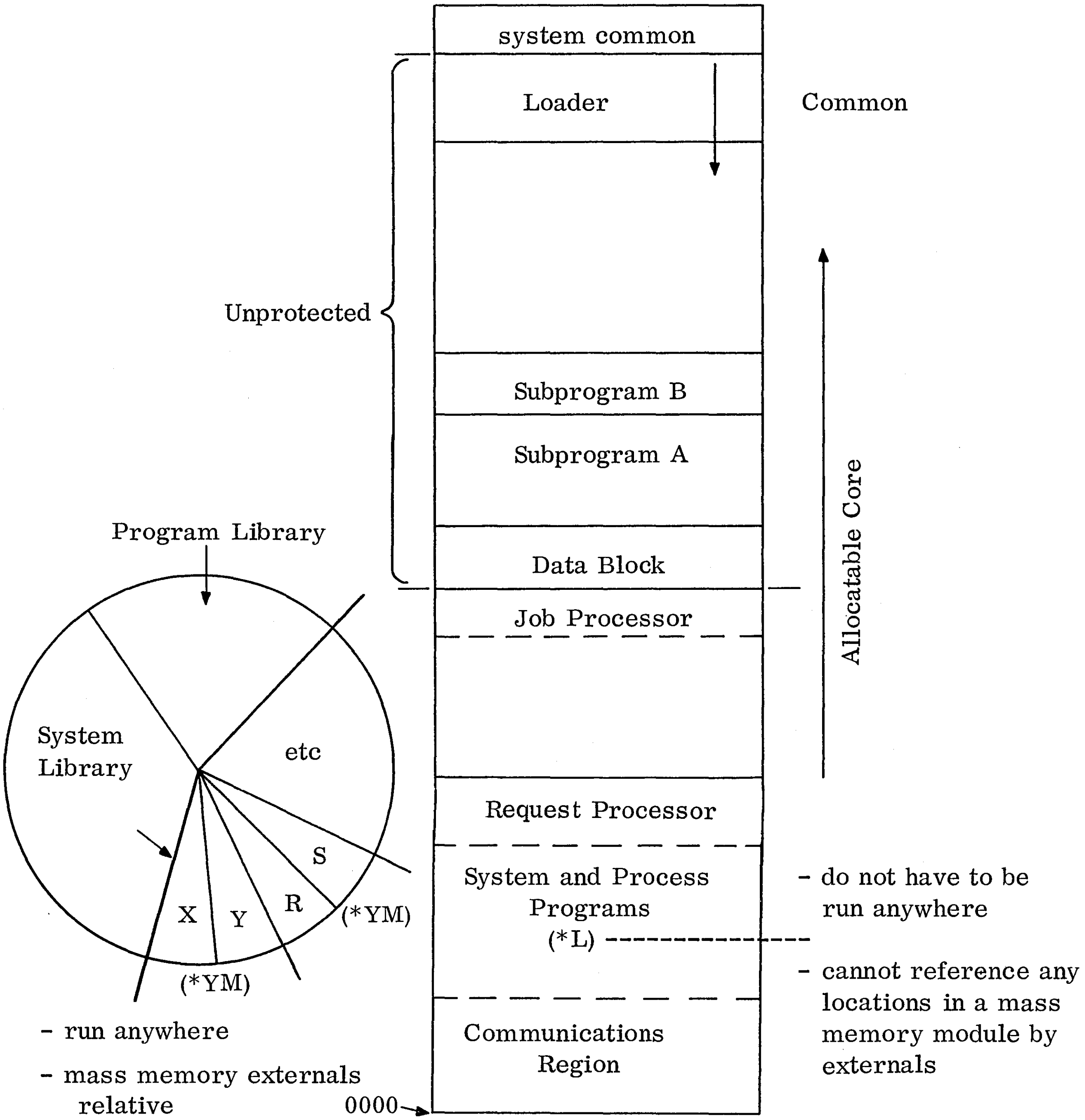
Core resident programs can make references to other core resident programs either relatively or absolutely. Since there is a choice, it is probably best to use absolute mode. Then if this program is ever made mass memory resident, its references to low core programs will be correct.

Core resident programs would not make any references to mass memory programs by way of externals. They would schedule the mass memory programs (which would be in the system directory).

Mass memory programs are grouped together in modules. A module may be composed of a single program or several programs. Externals which reference core resident programs must be absolute. Externals which reference other programs in the same module must be relative. Mass memory programs may not reference any programs in another module by externals (they can schedule them in).

A module has a name unique to it which is not an entry point anywhere in the system. This is the name in the system directory. Core resident programs can also be in the directory but only a few usually are. Any program which may at some later date be changed from core resident to mass memory resident (or vice versa) should be in the directory so that it can be scheduled by the system directory schedule request form.

Figure 28. 1700 Core Map



- do not have to be run anywhere
- cannot reference any locations in a mass memory module by externals

- run anywhere
- mass memory externals relative
- locore externals absolute
- cannot reference any locations in other mass memory modules by way of externals

Each "group" of mass memory programs loaded and linked together by one *YM control statement is called a mass memory module and has a name unique to the module.

CHAPTER XII

PERIPHERAL PROGRAMMING - I

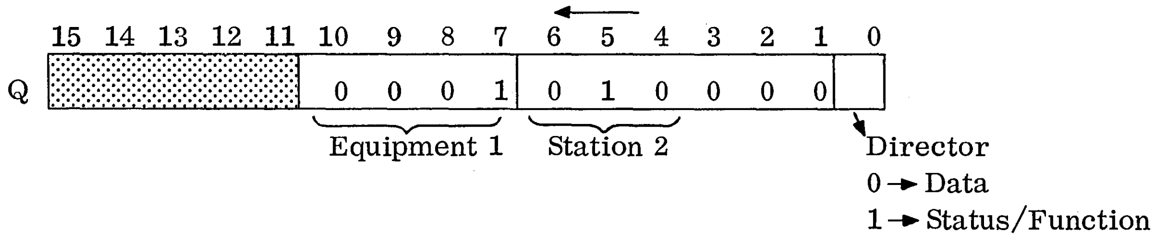
12

CHAPTER XII - Peripheral Programming I

12.1 1721 PAPER TAPE READER

The paper tape reader transcribes data to the lower 8 bits of the A Register at a rate of 350 eight-bit characters per second. The time between frames is 2.857 milliseconds. These times qualify the 1721 to be grouped with the low speed package; equipment number 1, station number 2.

The Q Register will be in the following format when referencing the paper tape reader.



The Q Register will contain either $00A0 or $00A1.

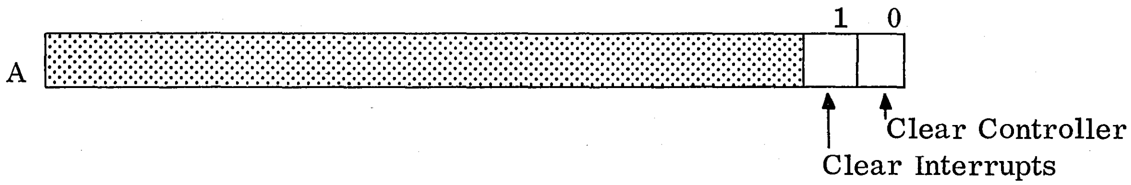
```
LDQ          =N$00A0          DATA
LDQ          =N$00A1          FUNCTION/STATUS
```

12.1.1 PTR Functions

The D portion dictates the operation to be performed in conjunction with the INP and OUT instructions.

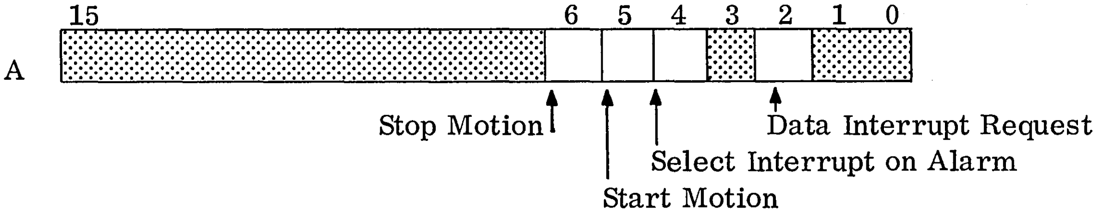
D = 1 FUNCTION (OUT)

The programmer may issue the functions together with the exception of the clear controller and clear interrupt functions which must be issued separately.



```
LDQ          =N$00A1          PTR for FUNC
ENA           3               CLR INT, CLR CONT
OUT          -1
```

The remaining functions may be issued jointly.



Data Interrupt -

Bit 2 allows the programmer to instruct the 1721 to interrupt the 1700 when the holding register on the 1721 contains a frame of data ready for transfer.

Alarm Interrupt -

Bit 4 permits the paper tape reader to interrupt the computer if one of the following conditions arise:

a) Paper Motion failure
b) Lost data
c) Power off

Start Motion -

Bit 5 starts the paper tape reader moving tape through the read station. Paper will continue to be moved through the read station until the motion is stopped or the reader runs out of paper.

Stop Motion -

Bit 6 stops the movement of the paper through the read station. The brakes on the 1721 assure stopping before reaching the next frame.

```
LDQ      =N$00A1      PTR for FUNC
LDA      =NFUNC       PLACE FUNCTION IN A
NOP
OUT      -1           INITIATE DESIRED OPERATIONS
```

The logic has been set up, therefore, the programmer needs only to bring the data into the computer.

D = 0 DATA

```
LDQ      =N$00A0      PTR FOR DATA
NOP
INF      -1           DATA IN LOW 8 BITS OF A
```

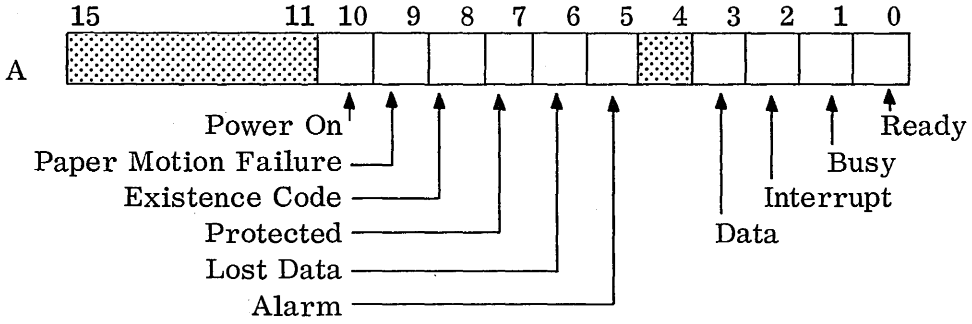
12.1.2 PTR Status

The programmer may monitor the operations of the paper tape reader by taking status.

D = 1 STATUS (INP)

```
        LDQ        =N$00A1          PTR FOR STATUS
        NOP
        INP        -1               STATUS IN A
```

The status bits are in the A Register.



Ready (Bit 0): Power is on and paper tape has been loaded into the reader. The preparations have been made known to the logic by pressing the READY switch on the paper tape reader console. The reader becomes Not Ready if a paper motion failure occurs or if the power is turned off.

Busy (Bit 1): The paper tape reader is Busy if a Start Motion command has been issued and no Stop Motion command has followed. Motion stops on a Stop Motion command, a paper motion failure, or if the power is turned off.

Interrupt (Bit 2): An interrupt condition exists. Other status bits must be examined to determine the condition causing this interrupt.

Data (Bit 3): The Data Hold register in the paper tape reader contains an 8-bit frame of data which is ready for transfer to the computer. Start Motion must be set to receive this status. The status drops when the Data Hold register is emptied by transfer to the computer.

Alarm (Bit 5): At least one of the following conditions exists in the paper tape reader: (1) paper motion failure (bit A9), (2) lost data (bit A6), or (3) power off (bit A10 is '0').

Lost Data (Bit 6): When in interrupt on Data mode, paper motion continues after the Data Hold register is full. If the data is not transferred to the computer before the next frame appears, a lost data status occurs to show a frame has been passed. The time between frames is 2.857 milliseconds. The status drops when a clear controller command is sent. Lost data stops tape motion.

Protected (Bit 7): The PROGRAM PROTECT switch is on. This switch on the paper tape reader works in conjunction with the PROGRAM PROTECT switch on the computer. If the switch on the computer is off and the PROGRAM PROTECT switch of the peripheral device is on, no action is taken but the status bit is set to indicate the switch is on. If the switch on the computer is set, all rules of program protection apply. The paper tape reader in this condition only accepts protected instructions.

Existence Code (Bit 8): The paper tape reader is attached. If the bit is a '1', the reader is missing from the particular computer system.

Paper Motion Failure (Bit 9): No change in the feed hole circuit has occurred for 40 milliseconds while trying to read. The paper motion failure causes the reader to become Not Ready; it can only be made ready by pushing the READY switch or by a Clear Controller command. It is considered an illegal operation to send any other function code to the reader or a read command until the READY switch has been pressed or a Clear Controller has been issued.

Power On (Bit 10): Power to the reader is on. If this bit is a '0', power is off.

12.1.3 Example

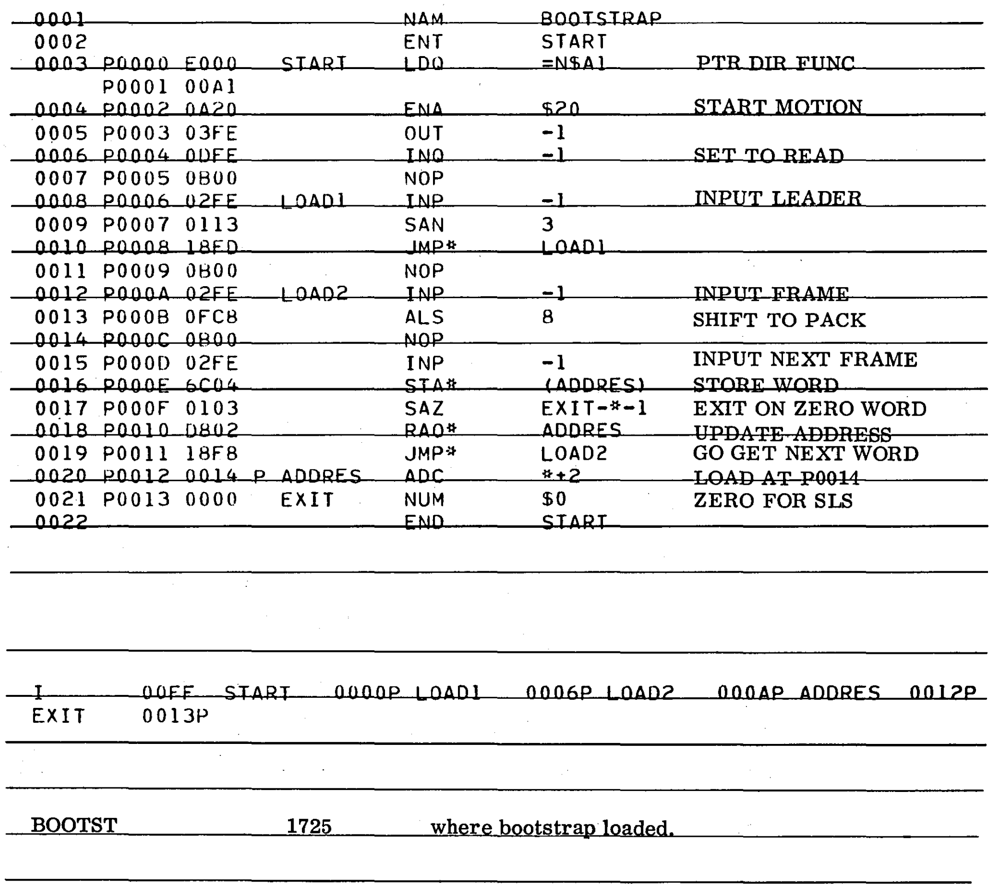
The following is a test program for the 1721 paper tape reader. It inputs data, beginning with the first nonzero frame, until a zero word is encountered, at which time data input stops. The data is stored in the consecutive locations beginning after the end of the program.

The controller should be cleared from the console before the program begins, as the clear controller function cannot be sent with the start motion function at line 4. The stop switch should be set so the program will stop after reading the tape. After it stops, if the switch is set to RUN, the tape which was just read will execute (assuming it contained an absolutized program).

1721 PTR - AUG '68
USDA

*CLEAR CONTROLLER FROM CONSOLE - CANNOT START MOTION & CLEAR CONTROLLER IN SAME FUNCTION

The following is a test routine for the 1711 teletype. It outputs an 11-word message from the buffer GET. The controller is cleared in a separate function before a new function is selected. Write mode is selected because Read mode is in effect after the clear controller function.

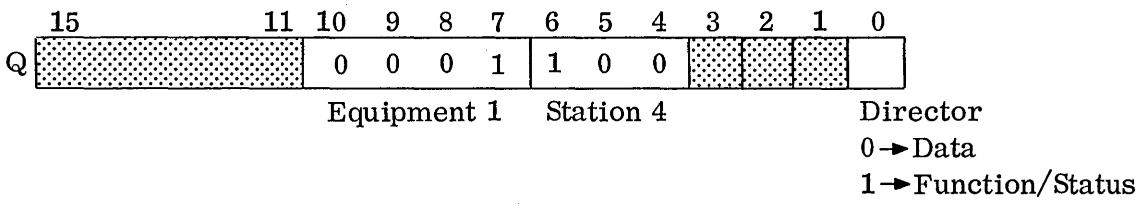


```
0001                             NAM      BOOTSTRAP
0002                             ENT      START
0003 P0000 E000     START        LDQ      =N$A1         PTR DIR FUNC
     P0001 00A1
0004 P0002 0A20                  ENA      $20           START MOTION
0005 P0003 03FE                  OUT      -1
0006 P0004 0DFE                  INQ      -1            SET TO READ
0007 P0005 0B00                  NOP
0008 P0006 02FE     LOAD1        INP      -1            INPUT LEADER
0009 P0007 0113                  SAN      3
0010 P0008 18FD                  JMP*     LOAD1
0011 P0009 0B00                  NOP
0012 P000A 02FE     LOAD2        INP      -1            INPUT FRAME
0013 P000B 0FC8                  ALS      8             SHIFT TO PACK
0014 P000C 0B00                  NOP
0015 P000D 02FE                  INP      -1            INPUT NEXT FRAME
0016 P000E 6C04                  STA*     (ADDRES)      STORE WORD
0017 P000F 0103                  SAZ      EXIT-*-1      EXIT ON ZERO WORD
0018 P0010 D802                  RAO*     ADDRES        UPDATE ADDRESS
0019 P0011 18F8                  JMP*     LOAD2         GO GET NEXT WORD
0020 P0012 0014 P ADDRES         ADC      *+2           LOAD AT P0014
0021 P0013 0000     EXIT         NUM      $0            ZERO FOR SLS
0022                             END      START


I       00FF  START   0000P LOAD1   0006P LOAD2   000AP ADDRES   0012P
EXIT    0013P


BOOTST              1725        where bootstrap loaded.
```

12.2 1723 PAPER TAPE PUNCH

The punch is grouped with Control Data's low speed package. The punch is a character device. It accepts the lower 8 bits of the A Register as data. These 8 bits may be ASCII codes or binary.

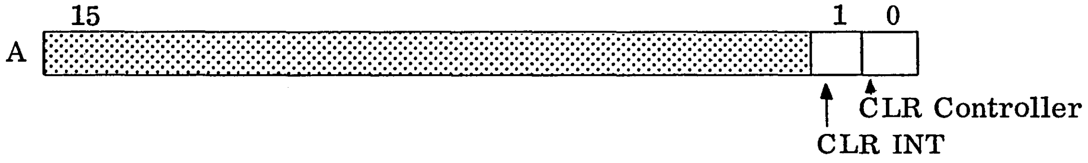
The punch is addressed as Equipment 1, Station 4.



```
LDQ        =N$00C0        PTP, DATA
LDQ        =N$00C1        PTP, FUNC/STATUS
```

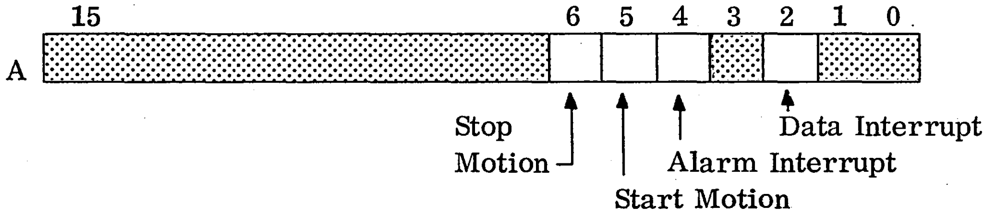
12.2.1 PTP Functions

The programmer may direct the operations and interrupt selections of the Punch via the Director function. The clear interrupt and clear controller function may be issued together but should not be issued with the other functions.



```
LDQ        =N$00C1        PTP, FUNC
ENA        3
OUT        -1
```

The remaining functions may be issued together.

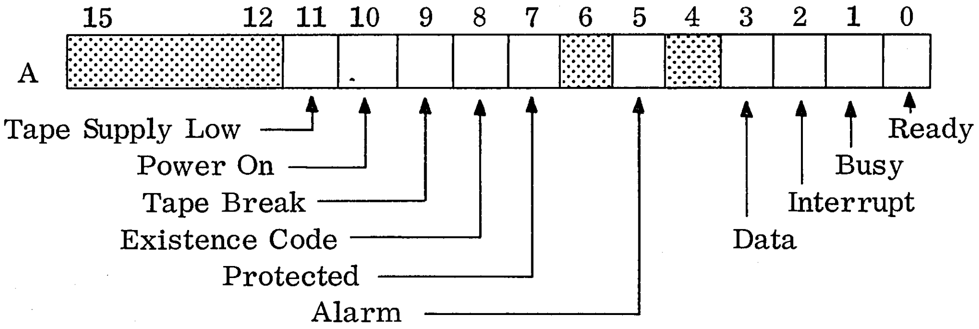


```
LDQ        =N$00C1        PTP, FUNC
LDA        =NFUNC         FUNCTIONS
OUT        -1             INITIATE FUNCTIONS
```

12.2.2 PTP Status

The programmer may status the Paper Tape Punch by setting the Director bit to a '1' and issuing an INP instruction.

```
LDQ        =N$00C1        PTP, STATUS
NOP
INP        -1             STATUS TO A
```



Ready (Bit 0): The paper tape punch is Ready when its power is on, tape has been loaded, and the READY switch on the station console has been pressed. The punch becomes NOT READY if tape break occurs or if power is turned off.

Busy (Bit 1): The punch is Busy if a Start Motion is in effect or until the punch has finished processing the data in the Data Hold register.

Interrupt (Bit 2): An interrupt condition exists. Other bits can be monitored to determine if one or more of the selected interrupts has occurred.

Data (Bit 3): The 8 bits of data in the Data Hold register of the punch have been punched and the new data may be received from the computer. The data status drops when a transfer from the computer is made.

Alarm (Bit 5): This status indicates that one of the following conditions has arisen:

a. Tape Break
b. Power off
c. Tape low

The status drops when the condition is corrected.

Protected (Bit 7): The PROGRAM PROTECT switch on the peripheral equipment is set. The status bit only indicates that the switch is set; it does not show if a program protect violation occurred. If the PROGRAM PROTECT switch on the computer is on, the punch does not accept commands which are not protected. All rules of program protection apply.

Existence Code (Bit 8): A zero setting acknowledges that the paper tape punch is attached. If the bit is a '1', the punch is missing from the particular computer system.

Tape Break (Bit 9): The tape break status bit is set if the punch supply tape has broken or run out and approximately 2 inches of tape remain. If the tape supply low bit is ignored, it results eventually in the Tape Break condition as the supply of tape is exhausted. The Tape Break condition causes the punch to become Not Ready. It can only be made Ready by loading paper tape and pressing the READY switch. However, it is still able to receive the Clear Controller and Clear Interrupts function codes so that the Interrupt signal (if Interrupt on Alarm was selected) can be dropped. It is considered an illegal operation to send any other function code or a Write signal until the READY switch has been pressed.

Power On (Bit 10): The power to the punch is on. If this bit is not a '1', the power is off and an Alarm interrupt may be generated.

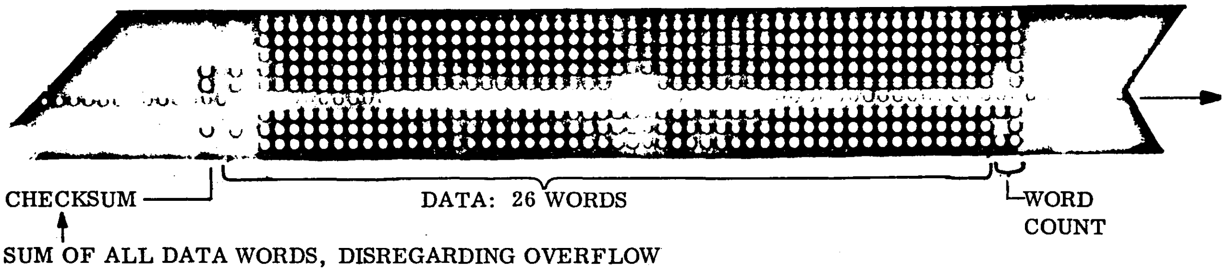
Tape Supply Low (Bit 11): The available supply of tape remaining to be punched is limited.

12.2.3 PTP Example Program

```
0001                                  NAM     PUNCH
0002                                  ENT     START
0003   P0000  0019                    BZS     AA(25) ←——————  SET BUFFER TO 1'S
0004          001A  P                 EQU     BB(AA+25)←         FROM CONSOLE
0005          00EA                    EQU     ADISP($FA)      NOTE THAT 26 WORDS WILL
0006          00C1                    EQU     PUNCH($00C1)       BE OUTPUT, NOT 25
0007                                  EXT     ERROR,DISP
0008   P0019  0002                    BSS     TEMP(2)         RESERVED FOR BLOCK SIZE
0009   P0018  0001                    BZS     CKS(1)
0010   P001C  C000     START          LDA     =XAA            FWA
       P001D  0000  P
0011   P001E  9000                    SUB     =XBB            LWA
       P001F  001A  P
0012   P0020  0113                    SAN     OK-*-1          IF UNEQUAL, ALL IS FINE
0013   P0021  1400  X                 JMP+    ERROR           CONTAINS SIZE OF BLOCK
       P0022  7FFF  X
0014   P0023  0B00                    NOP
0015   P0024  68F5     OK             STA*    TEMP+1          BLOCK SIZE COMPL
0016   P0025  0864                    TCA     A               COMPLEMENT BLOCK SIZE
0017   P0026  68F2                    STA*    TEMP            BLOCK SIZE A
0018   P0027  C000                    LDA     =X$7FFF-50
       P0028  7FCD
0019   P0029  60FF                    STA-    I               I EQUALS COUNTER FOR LEADER
0020   P002A  E000                    LDQ     =XPUNCH         PREPARE PUNCH TO RECEIV
       P002B  00C1
0021   P002C  0A01                    ENA     +1              CLEAR CONTROLLER
0022   P002D  03FE                    OUT     -1
0023   P002E  0A20                    ENA     $20
0024   P002F  03FE                    OUT     -1
0025   P0030  0DFE                    INQ     -1              PREPARE PUNCH FOR DATA
0026   P0031  0844                    CLR     A
0027   P0032  03FE     LOOP1          OUT     -1              JUMP ON SELF UNTIL OUT
0028   P0033  D0FF                    RAO-    I               UPDATE LEADER COUNT
0029   P0034  01A1                    SOV     NEXT-*-1           WHEN I=8000, 50 BLANKS
0030   P0035  18FC                    JMP*    LOOP1           KEEP OUTPUTTING BLANK L
0031   P0036  C8E3     NEXT           LDA*    TEMP+1          COMPLEMENT OF BLOCK SIZE
0032   P0037  0FC8                    ALS     8               HIGH ORDER BITS PUNCHED FIRST
0033   P0038  0B00                    NOP
0034   P0039  03FE                    OUT     -1
0035   P003A  0FC8                    ALS     8               LOW ORDER BITS SECOND
0036   P003B  0B00                    NOP
0037   P003C  03FE                    OUT     -1              SIZE OF BLOCK NOW ON TAPE
0038   P003D  0844                    CLR     A
0039   P003E  60FF                    STA-    I               ZERO OUT THE INDEX I
0040   P003F  C500     LOOP2          LDA+    AA,I            PLACE FIRST WORD OF DATA IN
       P0040  0000  P
0041   P0041  88D9                   ⎧ADD*    CKS             CHECKSUM
0042   P0042  68D8                   ⎩STA*    CKS
```

```
0043   P0043  C500               LDA+     AA,I             ORIGINAL DATA IN A
       P0044  0000  P
0044   P0045  0FC8               ALS 8
0045   P0046  0B00               NOP
0046   P0047  03FE               OUT      -1               OUTPUT FIRST CH.
0047   P0048  0FC8               ALS      8
0048   P0049  0800               NOP

                    Could
0049   P004A  03FE  compute ---> OUT      -1               OUTPUT SECOND CH.
0050   P004B  D0FF  CKSUM        RAO-     I                UPDATE INDEX
0051   P004C  C0FF  here         LDA-     I                PLACE UPDATE IN A
0052   P004D  98CB               SUB*     TEMP
0053   P004E  0101               SAZ      SUM-*-1          BLOCK IS PUNCHED
0054   P004F  18EF               JMP*     LOOP2
0055   P0050  C8CA  SUM          LDA*     CKS
0056   P0051  0FC8               ALS      8
0057   P0052  0B00               NOP
0058   P0053  03FE               OUT      -1               FIRST CH. OUT
0059   P0054  0FC8               ALS      8
0060   P0055  0B00               NOP
0061   P0056  03FE  sov 0 -----> OUT      -1               CHECKSUM ON PAPER TAPE
0062   P0057  C000               LDA      =X$7FFF-50       PREPARE FOR END LEAD
       P0058  7FCD
0063   P0059  60FF               STA-     I                INITIALIZE INDEX I
0064   P005A  0844               CLR      A
0065   P005B  68BF               STA*     CKS
0066   P005C  03FE  LOOP3        OUT      -1               OUTPUT BLANK LEADER
0067   P005D  D0FF               RAO-     I                UPDATE INDEX
0068   P005E  01A1               SOV      DONE-*-1            IF OVERFLOW FINISHED
0069   P005F  18FC               JMP*     LOOP3
0070   P0060  0D01  DONE         INQ      1                PREPARE PTP FOR FUNCTION
0071   P0061  0A01               ENA      1
0072   P0062  03FE               OUT      -1               CLEAR CONTROLLER
0073   P0063  C000               LDA      =N$40            STOP MOTION
       P0064  C040
0074   P0065  03FE               OUT      -1               OUTPUT FUNCTION
0075   P0066  14EA               JMP-     (ADISP)
0076                             END      START
```

```
I         00FF    START    001CP   AA       0000P  BB       001AP  ADISP    00EA
PUNCH     00C1    TEMP     0019P   CKS      001BP  OK       0024P  LOOP1    0032P
NEXT      0036P   LOOP2    003FP   SUM      0050P  LOOP3    005CP  DONE     0060P
DISP      7FFFX   ERROR    0022X
```

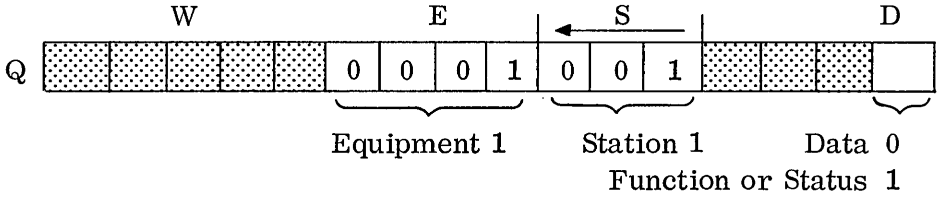


12.3 1711 TELETYPEWRITER

The 1711 Teletype may send and receive information. The data transmission to or from the 1711 takes 100 milliseconds.

The Teletype is one of three devices composing the low speed package which is always Equipment Number 1. All interrupts generated by the teletype shall be processed via line 1 interrupts.

The Q Register will contain $0091 or $0090 when communicating with the teletype. Break this word down into Binary, and we have Equipment 1, Station 1.



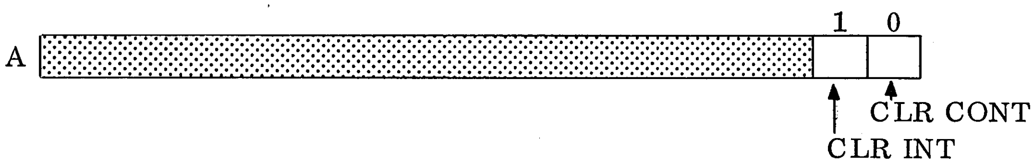
The D portion denotes the transfer of data when set to zero. All data transmissions will be to and from the lower 8 bits of the A Register. The characters will be transferred in ASCII codes, one character at a time. The directional flow of the data will be governed by the INP and OUT instructions.

```
LDQ        =N$0090        TTY FOR DATA
NOP
INP        -1             READ DATA INTO A

LDQ        =N$0090        TTY FOR DATA
LDA        BUF            PUT DATA IN A
NOP
OUT        -1             WRITE DATA ON TTY
```

12.3.1 TTY Functions

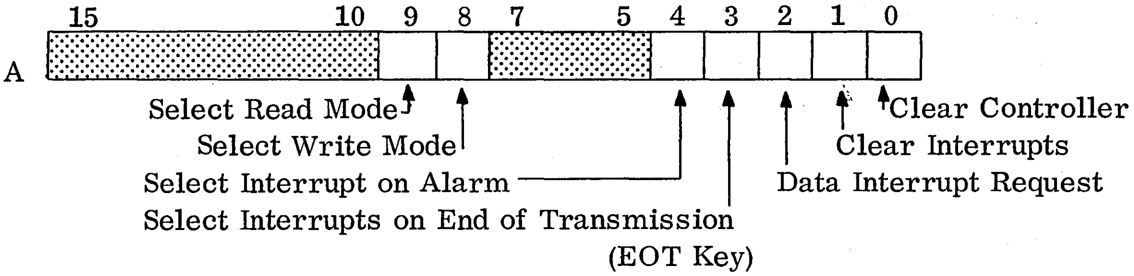
The D portion indicates a function or a status when set to a 1. A function is indicated by issuing an OUT instruction with A preset to the function.

```
LDQ        =N$0091        TTY FOR FUNC OR STATUS
LDA        FUNC           PLACE FUNCTION IN A
NOP
OUT        -1             OUTPUT A FUNCTION
```



The clear controller and clear interrupt functions must be sent to the teletype prior to selecting other functions.

12.3.1



The 1711 provides the capability of selecting three interrupts; DATA, ALARM, and END OF TRANSMISSION (EOT). The DATA interrupt will occur when the teletype is prepared to send or receive data. The ALARM interrupt will be generated if data has been lost or if the teletype goes from READY to NOT READY. The END OF TRANSMISSION interrupt will occur whenever the EOT key on the console of the teletype has been pressed. None of these conditions will generate an interrupt unless the programmer has selected them.

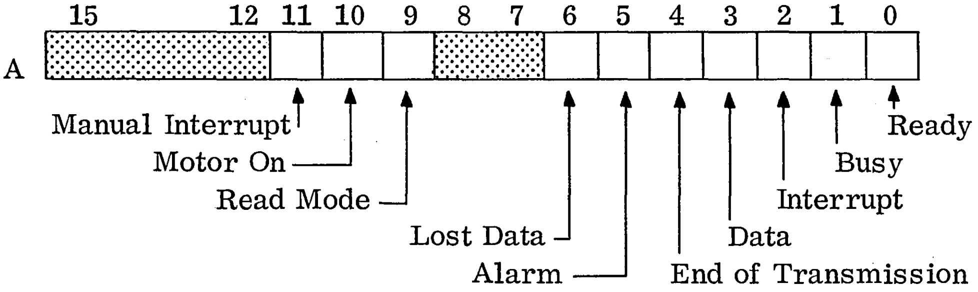
The motor will be started on the 1711 by an output of a dummy character - to the teletype.

12.3.2 TTY Status

Status will be sent to the A Register from the teletype with an INP instruction.

```
LDQ       =N$0091       TTY FOR STATUS OR FUNC
NOP
INP       -1            INPUT STATUS TO A
```

The A Register will contain the status and the programmer may examine it to determine the next procedure he wishes to follow.



Ready (A0 is 1): If this bit is set in the A Register, the Power switch on the console of the teletypewriter is in the ON-LINE position and the motor is on.

Busy (A1 is 1): If this bit is set, one or more of the following conditions exist:

a) The controller is in Read mode and is in the process of receiving a character from the teletypewriter or the Data Hold register contains data for transfer to the computer. The Busy status drops upon completion of the transfer to the computer if data has not been lost. If data has been lost, the controller requires 200 milliseconds to stop the teletypewriter and remains Busy all this time.

b) Write mode and the Data Hold register contains data and is in the process of transferring it to the teletypewriter. Busy drops upon completion of the transfer.

c) Either mode and the controller is in the process of starting the motor in the teletypewriter. In Write mode output of a character starts the motor and this character is lost. In Read mode, the BREAK key must be pressed to start the motor.

Interrupt (A2 is 1): An interrupt condition exists. Other bits must be monitored to determine the condition causing this interrupt.

Data (A3 is 1): An interrupt is generated and this status bit is a 1 under the following conditions:

a) Read mode and the Data Hold register contains data for transfer to the computer. The status drops upon completion of a Read.

b) Write mode and the controller is ready to accept another Write from the computer. The status drops upon completion of the Write.

End of Transmission (A4 is 1): The Data Hold register contains the End of Transmission code. This code is generated by pressing the EOT key on the keyboard of the teletypewriter. The end of transmission status drops upon the completion of the next Write or Read.

Alarm (A5 is 1): The teletypewriter is not in a Ready state or has lost data.

Lost Data (A6 is 1): The controller was not serviced by the computer before a new character was sent by the teletypewriter. The keyboard and tape transmitter are locked out. The status bit indicates a Lost Data condition, and a Busy status indicates that the process of stopping the teletypewriter is in progress. Data held in the Data Hold register is not disturbed, but the incoming data is ignored. The lost data status canbe cleared by a Clear Controller or a Select Write Mode command. These two functions are rejected while the controller is stopping the teletypewriter. The Select Write mode command must be preceded by a Read operation to clear the Data Hold register. After the teletypewriter has stopped, the computer may do an Output operation to notify the controller of the Error condition.

Read Mode (A9 is 1): If this bit is a 1, the controller is conditioned for an Input operation from the teletypewriter. Read mode is automatically in effect after a clear controller function has been issued.

Motor On (A10 is 1): The motor of the teletypewriter is on. The presence of this bit indicates that the teletypewriter motor is on and up to speed.

a) Write mode: Motor starts with the output of a character. Two-second delay occurs between output of the character and this status bit being set to allow the motor to get up to speed.

b) Read mode: Press the BREAK key to turn on the motor. Two-second delay also occurs between the action of the BREAK key and the status bit being set to allow the motor to get up to speed.

Manual Interrupt (A11 is 1): The manual interrupt button on the teletype has been pressed.

Example:

The following is a test routine for the 1711 teletype. It outputs an 11-word message from the buffer GET. The controller is cleared in a separate function before a new function is selected. Write mode is selected because Read mode is in effect after the clear controller function.

12.3.3 TTY Example Program

```
0001                              NAM       TYPE OUT
0002 P0000 504C    GET            ALF       11,PLEASE INPUT YOUR CODE
     P0001 4541
     P0002 5345                                                 Message
     P0003 2049
     P0004 4E50
     P0005 5554
     P0006 2059
     P0007 4F55
     P0008 5220
     P0009 434F
     P000A 4445
0003 P000B 0844                   CLR       A
0004 P000C 60FF                   STA-      I
0005 P000D E000                   LDQ       =N$0091          TTY DIR FUNC
     P000E 0091
0006 P000F 0A03                   ENA       $3               CLR CONTR & INT
0007 P0010 03FE                   OUT       -1
0008 P0011 C000                   LDA       =N$100           WRITE MODE
     P0012 0100
0009 P0013 0B00                   NOP
0010 P0014 03FE                   OUT       -1
0011 P0015 0DFE                   INQ       -1               WRITE DATA FUNC
0012 P0016 0B00                   NOP
0013 P0017 03FE                   OUT       -1               SEND DUMMY CHAR
0014 P0018 C9E7    LOOP           LDA*      GET,I            OUTPUT DATA
0015 P0019 0FC8                   ALS       8
0016 P001A 0B00                   NOP
0017 P001B 03FE                   OUT       -1
0018 P001C 0FC8                   ALS       8
0019 P001D 0B00                   NOP
0020 P001E 03FE                   OUT       -1
0021 P001F D0FF                   RAO-      I
0022 P0020 C0FF                   LDA-      I
0023 P0021 09F4                   INA       -11
0024 P0022 0101                   SAZ       DONE-*-1
0025 P0023 18F4                   JMP*      LOOP
0026 P0024 0000    DONE           0         0
0027                              END


 I        00FF  GET       0000P LOOP      0018P DONE      0024P
```

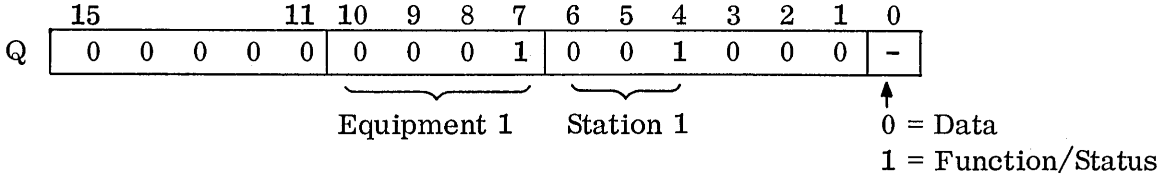
12.3.3

Typewriter Printout

```
PP
*
MI
*K, I5
J
*P
J
*ASSEM
L, 03  FAILED 01
ACTION
CU
L, 03 FAILED 01
ACTION
CU
L, 03 FAILED 01
ACTION
CU
J
*P
J
*L, 8                                  Output from program
J
PLEASE INPUT YOUR CODE
```

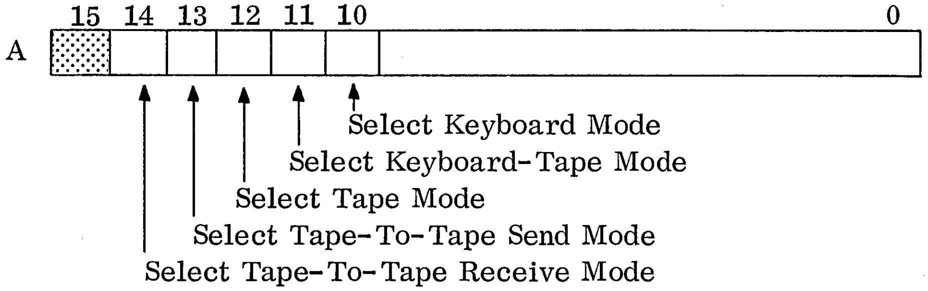
12.4 1713 TELETYPEWRITER

The 1713 is composed of a keyboard, printer, paper tape reader, and a paper tape punch, each accessible by the computer. The 1713 is grouped with the low speed package, equipment number 1, station number 1. The Q Register will be in the following format when referencing the 1713.



```
LDQ        =N$0090        SEL TTY, DATA
LDQ        =N$0091        SEL TTY, STATUS OR FUNCTION
```

The programmer has the option of selecting the reader, punch, printer or keyboard. All four units may be used together or separately. The selection is made by setting an appropriate bit in the function word.


15
14
13
12
11
10
0
A
Select Keyboard Mode
Select Keyboard-Tape Mode
Select Tape Mode
Select Tape-To-Tape Send Mode
Select Tape-To-Tape Receive Mode

Bit 14 ⟶ Connects the punch to the controller leaving the keyboard and reader inactive.

Bit 13 ⟶ Connects the reader only to the controller.

Bit 12 ⟶ Connects the page printer and reader to the controller. The keyboard and punch are connected together as an off-line tape preparation device. Read operations transfer information from the paper tape reader to the controller and the page printer. Write operations transfer information to the page printer. Simultaneously, a new tape can be prepared from keyboard entries.

Bit 11 ⟶ Connects the keyboard, page printer, reader and punch to the controller. A character struck on the keyboard or sent from the reader is printed, punched and transmitted. A character sent to the 1713 is printed and punched.

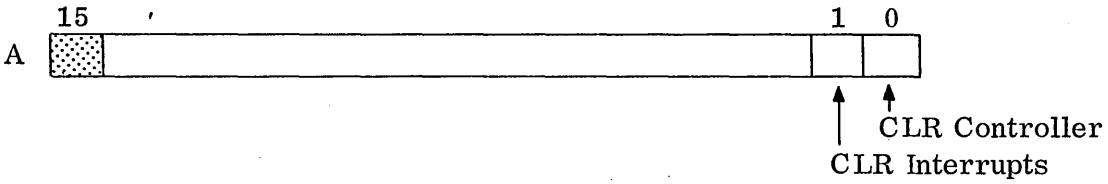
Bit 10 ⟶ Connects the keyboard and printer to the controller which act as a send/receive page printer. The paper tape units are inactive in this mode.

12.4

The 1713 accepts the lower eight bits of the A Register as data and sends eight bits of data to the A Register. All codes going to the page printer must be eight bit ASCII codes. The data transfer rate is 100 milliseconds per character.

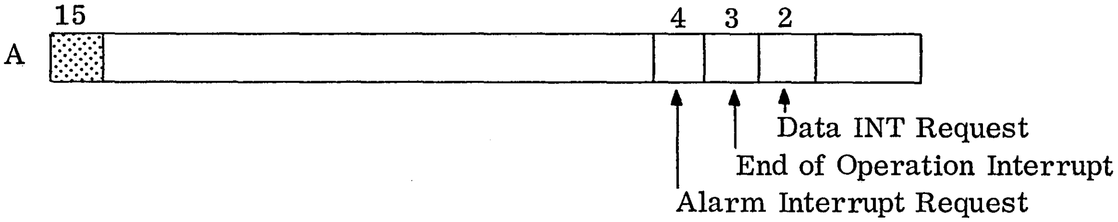
12.4.1 1713 Functions

Prior to selecting a mode, the controller and interrupts may be cleared by issuing a function to the 1713.



```
LDQ         =N$91          SEL TTY, FUNCTION
ENA         $3             CLR CONT & CLR INT
OUT         -1
```

Interrupts may be selected by setting the following bits.



Bit 2 ——→ Allows the 1713 to interrupt the computer whenever the holding register is ready to send or receive data.

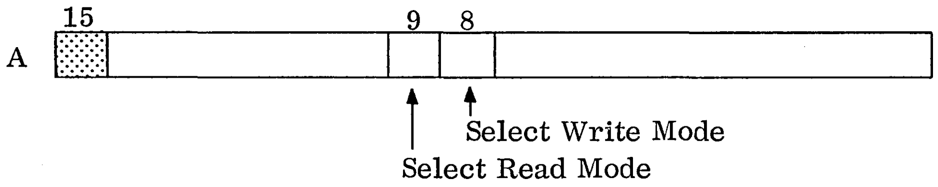
Bit 3 ——→ Notifies the 1713 to interrupt the computer whenever an operation is completed.

Bit 4 ——→ Provides for an interrupt whenever an alarm condition arises. Alarm conditions:

1. 1713 becomes NOT READY
2. LOST DATA
3. Out of Tape

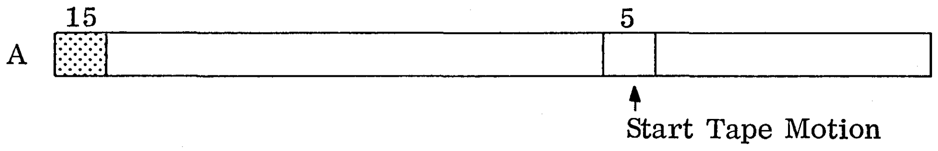
```
LDQ         =N$91          SEL TTY, FUNCTION
LDA         =N$1C          INT ON DATA, ALARM, EOP
NOP
OUT         -1
```

The operating mode, READ or WRITE, is selected by setting a bit in the function word.



```
LDQ     =N$91      SEL TTY, FUNCTION
LDQ     =N$100     WRITE MODE
NOP
OUT     -1

LDQ     =N$91      SEL TTY, FUNCTION
LDA     =N$200     READ MODE
NOP
OUT     -1
```

When running in Tape-To-Tape Send Mode, the program must issue a START TAPE MOTION function. When reading data from the reader a start motion function must be issued after each character.



```
LDQ     =N$91      SEL TTY, FUNCTION
LDA     =N$2220    TAPE SEND, READ, START TAPE
                   MOTION
NOP
OUT     -1
```

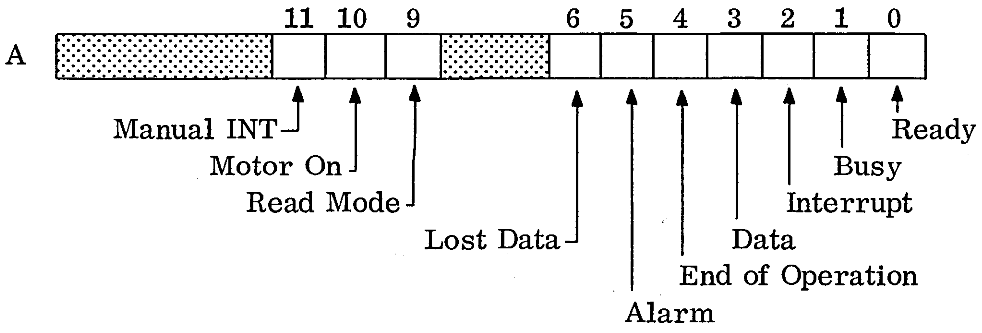
All of the functions mav be issued together with the exception of the clear controller and clear interrupt functions which must be issued separately.

12.4.2 1713 Status

Status may be taken on the 1713 at any time.

```
LDQ     =N$91      SEL TTY, STATUS
NOP
INP     -1         STATUS IN A
```

12.4.2



When the corresponding bit is a 1, the condition exists.

READY → The 1713 is capable of performing operations and accepting functions.

BUSY → The 1713 is in the process of performing an operation.

INTERRUPT → An interrupt has been generated.

DATA → The holding register in the 1713 is prepared to send or receive data.

END OF OPERATION → The 1713 has completed an operation.

ALARM → An alarm condition exists in the 1713.

LOST DATA → The 1713 had data in the holding register which was not picked up by the computer before another character was read into the register.

READ MODE → The 1713 is in a read mode. If this bit were not set, the write mode would be in effect.

MOTOR ON → The 1713 motor is on. If this bit were not set, the motor is not on.

MANUAL INTERRUPT → The manual interrupt button on the 1713 has been pressed.

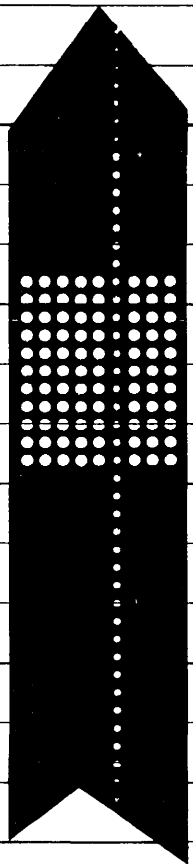
12.4.3 Example Program, 1713 TTY

The following program generates 10 frames of data in the A Register and punches them on paper tape. It then stops and waits for the tape to be put in the reader. It reads 20 frames and stores them in a buffer. Program read can be checked by sweeping the buffer.

The STOP Switch should be set before the program is run.

```
0001                          NAM      PT1713
0002                          ENT      PT1713
0003 P0000 0000   PT1713      0        0
0004 P0001 E000               LDQ      =N$91         TTY, FUNC/STATUS
     P0002 0091
0005 P0003 0A03           ENA          $3            CLR CONTR, CLR INT
0006 P0004 03FE               OUT      -1
0007 P0005 02FE               INP      -1            INP STATUS
0008 P0006 0FCF               ALS      15            READY BIT
0009 P0007 0131               SAM      1             SKIP WHEN READY
0010 P0008 18FC               JMP*     *-3           GO WAIT TILL READY
0011 P0009 C000               LDA      =N$4120       TTR, WRITE, START MOTION
     P000A 4120
0012 P000B 0B00               NOP
0013 P000C 03FE               OUT      -1
0014 P000D 0DFE               INQ      -1            PREPARE TO SEND DATA
0015 P000E 5841               RTJ*     LEADER        GO OUTPUT LEADER
0016 P000F 0A0F   WRITE       ENA      $F            DATA $F IN A
0017 P0010 03FE               OUT      -1            OUTPUT DATA
0018 P0011 D0FF               RAO-     I
0019 P0012 C100               LDA      =N-10,I       10 FRAMES
     P0013 FFF5
0020 P0014 0101               SAZ      1
0021 P0015 18F9               JMP*     WRITE
0022 P0016 5839               RTJ*     LEADER        GO OUTPUT TRAILER
0023 P0017 0000   STOP        SLS      0
0024 P0018 0D01               INQ      1             PREPARE FOR FUNCTION
0025 P0019 0A01               ENA      1             CLEAR CONTROLLER
0026 P001A 03FE               OUT      -1
0027 P001B 02FE               INP      -1            INPUT STATUS
0028 P001C 0FCF               ALS      15            READY BIT
0029 P001D 0131               SAM      1             SKIP WHEN READY
0030 P001E 18FC               JMP*     *-3           GO WAIT UNTIL READY
0031 P001F C000   SEL         LDA      =N$2000       TTS MODE
     P0020 2000
0032 P0021 0B00               NOP
0033 P0022 03FE               OUT      -1
0034 P0023 0A20   MOTION         ENA   $20           START MOTION
0035 P0024 03FE               OUT      -1
0036 P0025 0DFE               INQ      -1            PREPARE FOR DATA
0037 P0026 0B00               NOP
0038 P0027 02FE   LDR         INP      -1            INPUT DATA
0039 P0028 011B               SAN      STORE-*-1     STORE WHEN DATA COMES
0040 P0029 0D01               INQ      1
0041 P002A 18F4               JMP*     SEL           GO WAIT FOR DATA
0042 P002B C000   READ        LDA      =N$2000       TTS MODE
     P002C 2000
0043 P002D 0D01               INQ      1             FUNCTION
0044 P002E 0B00               NOP
0045 P002F 03FE               OUT      -1
0046 P0030 0A20               ENA      $20           START TAPE MOTION
0047 P0031 03FE               OUT      -1
0048 P0032 0DFE               INQ      -1            DATA
0049 P0033 02FE               INP      -1            INPUT DATA
0050 P0034 6907   STORE       STA*     BUF,I         STORE DATA
```

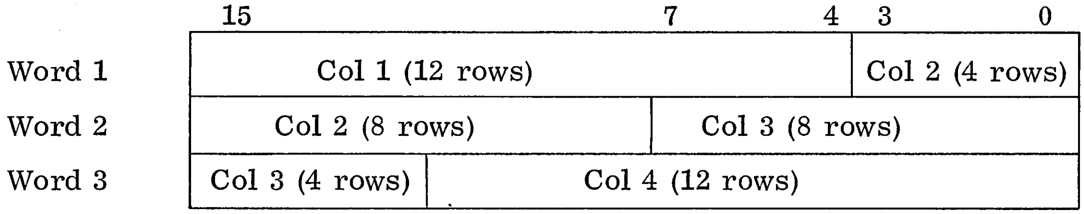
```
0051 P0035 D0FF                  RAO-      I
0052 P0036 C100                  LDA       =N-20,I        20 FRAMES
     P0037 FFEB
0053 P0038 0101                  SAZ       FINI-*-1
0054 P0039 18F1                  JMP*      READ
0055 P003A 0000     FINI         SLS       0              STOP AFTER READING
0056 P003B 0014     BUF          BZS       BUF(20)        SWEEP BUFFER TO CHECK
0057 P004F 0000     LEADER       0         0              WRITE LEADER OR TRAILER
0058 P0050 C000                  LDA       =X-50
     P0051 7FCD
0059 P0052 60FF                  STA-      I
0060 P0053 0844                  CLR       A
0061 P0054 03FE     OUTLDR       OUT       -1
0062 P0055 D0FF                  RAO-      I
0063 P0056 01A1                  SOV       1
0064 P0057 18FC                  JMP*      OUTLDR
0065 P0058 60FF                  STA-      I
0066 P0059 1CF5                  JMP*      (LEADER)
0067                             END
```



12.5 1726/405 CARD READER

The 405 is a Non-Buffered Card Reader capable of reading 1200 80-column cards per minute or 1600 51-column cards per minute. The transfer rate for one 80-column card is 384 microseconds.

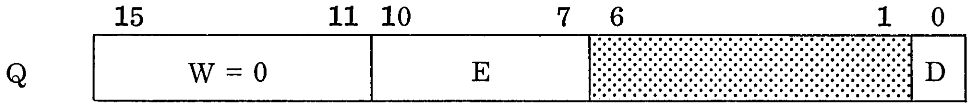
The twelve rows in each column constitute the 12-bit data word transferred to the computer. Software packing must be performed in order to form a 16-bit word. The format for the columns in relation to memory words is as follows:



Four card columns represent three computer words, therefore, one 80-column card represents 60 memory words.

The data read by the 405 may be buffered if connected to the 1706.

The Q Register will be in the following format when referencing the 405 Card Reader.

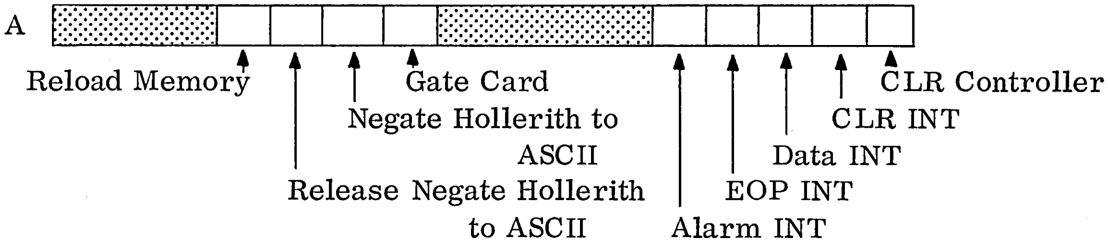


The E portion will correspond with the setting of the equipment switch on the controller. The D portion designates the operation to be performed.

12.5.1 CR Functions

D = 1 DIRECTOR FUNCTION (OUT)

```
LDQ     =N$0101     EQUIP 2, FUNC
LDA     =NFUNC      FUNCTION IN A
NOP
OUT     -1          ESTABLISH LOGIC
```



The functions for the 405 Card Reader may be issued jointly.

Clear Controller (Bit 0): Directs the clearing of all interrupt requests, motion requests, errors, and other logic that may be cleared. This function is subordinate to all other functions.

Clear Interrupts (Bit 1): Clears all interrupt requests and their responses.

Data Interrupt Request (Bit 2): Sets the interrupt request to be set which causes an interrupt to be generated when an information transfer may occur.

Interrupt on End of Operation (Bit 3): Requests an interrupt to be generated when the last card column has been read or a Reload Memory Function has been performed.

Interrupt on Alarm (Bit 4): Generates an interrupt whenever any of the following conditions arise:

1. Compare or pre-read error
2. Stacker full or jam
3. Input tray empty
4. Fail to feed
5. Separator card is read into computer memory
6. Auto/man switch is in man position

Gate Card (Bit 9): This bit gates the card being read to the secondary stacker. This function must be performed during the 1.5 milliseconds following the input of the last column to the buffer memory of the Card Reader.

Negate Hollerith to ASCII (Bit 10): When bit A10 is selected, 7 and 9 punch positions in column 1 are ignored and all information (binary or Hollerith) is read as binary. Bit A10 is subordinate to bit A11. Bit A10 is rejected if the controller is Busy.

NOTE

> Before beginning a new operation, make certain that bit A10 and the following bit, A11, are appropriately selected. If this is not done, the cards will be read in the mode or state that the card reader was in during the previous operation.

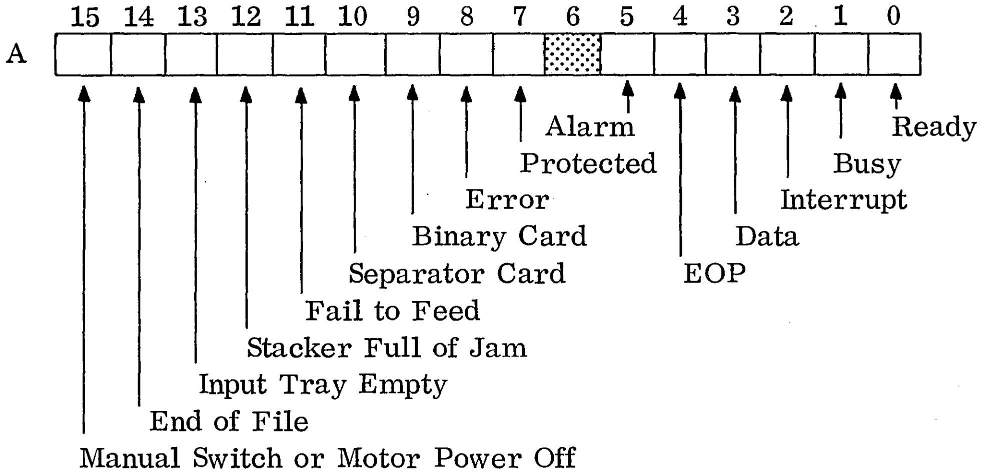
Release Negate Hollerith to ASCII (A11 = 1): When bit A11 is selected, the 7 and 9 punch positions in column 1 determine whether the card information is to be transferred in ASCII code or in binary form. The Release Negate Hollerith to ASCII function takes precedence over the select Negate Hollerith to ASCII function, and it is rejected if the controller is Busy. See Note.

Reload Memory (A12 = 1): This bit directs the controller to initiate a card feed thereby reloading the controller memory with the data from the next card in the card reader. The data that has not been transmitted from the memory to the computer is lost when Reload Memory is executed. A Reload Memory is required only if less than a full card of information is desired. Bit A12 is rejected if the controller is Busy.

12.5.2 CR Status

D = 1 STATUS (INP)

```
        LDQ        =N$0101        EQUIP 2, STATUS
        NOP
        INP        -1             STATUS IN A
```



Ready (Bit 0): The presence of this bit indicates that the card reader is ready for operation.

Busy (Bit 1): The controller is Busy whenever a card is being entered into the buffer memory.

Interrupt (Bit 2): The interrupt status is available if one or more of the selected interrupts has occurred. Other bits must be monitored to determine the condition causing the interrupt.

Data (Bit 3): This status bit indicates that data is ready to be transferred to the computer.

End of Operation (Bit 4): This status bit indicates that the last card column has been read from the buffer memory, or a reload memory function has been sent. This bit remains a 1 until a Reply signal is sent, or a Clear Controller function or Master Clear is issued.

Alarm (Bit 5): The bit remains a 1 until whatever caused the Alarm condition is removed. This status bit indicates that one or more of the following conditions has occurred:

1. Compare or Pre-read error
2. Stacker full or jam
3. Input tray empty
4. Fail to feed
5. A separator card has been transferred to the computer memory.
6. The AUTO/MAN switch is in the MAN position

Status bit A06 is not used.

Protected (Bit 7): This status bit indicates that the controller recognizes only the I/O instructions that have the protect bit present. This status bit is a 1 when the PROTECTED/UNPROTECTED switch is in the PROTECTED position.

Error (Bit 8): This bit indicates that a Pre-read or Compare error has occurred.

Binary Card (Bit 9): This bit is present when the contents of the first card column have been transferred to the computer memory and a binary card (rows 7 and 9 punched in first column) was detected, or the Negate Hollerith to ASCII function was selected. This bit remains a 1 until a Clear Controller or Master Clear function is issued, or a Reply is sent when a card is read under the following conditions:

1. The card is not a binary or a separator card.
2. The Release Negate Hollerith to ASCII function is selected.

Separator Card (Bit 10): This bit is present when the contents of the first card column have been transferred to computer memory and a separator card (rows 6, 7, 8, and 9 punched in first column) was detected. This bit remains a 1 until a Reply is sent when a card is read that is not a separator card, or until a Master Clear or Clear Controller function is executed.

Fail to Feed (Bit 11): This bit is a 1 if another card is not detected at the primary read station 500 ms after the previous card has cleared the secondary read station.

Stacker Full or Jam (Bit 12): This bit is a 1 when the stacker is full of cards or when the cards have jammed.

Input Tray Empty (Bit 13): This bit is a 1 when the input tray is empty.

End of File (Bit 14): This status bit becomes a 1 when the input tray is empty, the buffer memory is unloaded, and the END OF FILE switch is on. When the input tray does not contain the last card of a file, the switch should be off to inhibit this status bit.

Manual (Bit 15): This status bit is a 1 when the AUTO/MAN switch is in the MAN position or the MOTOR POWER switch is off.

D = 0 DATA (INP)

```
        LDQ       =N$0100       EQUIP 2, DATA
        NOP
        INP       -1            DATA IN A REG
```

	15			12	11	0
A	0	0	0	0	DATA	

Packing must be performed in order to obtain 16 bit words as indicated earlier in this discussion.

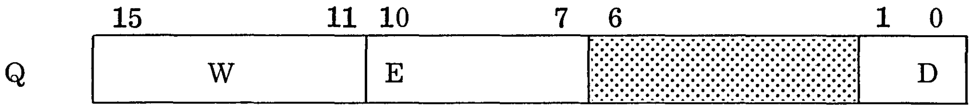
12.5.3 CR Example Program

```
                   NAM         CARDRD
                   EQU         AA($3FF),CARD($0281)    SEL EQUIP 5 F
                   EQU         MASKB($000F)
                   BSS         TEMP(2)
START              CLR         A
                   STA-        I
SECOND             LDQ         =XCARD
                   LDA         =N$1A03                 CLR CON CLR INT RM H TO A
                   NOP
                   OUT         -1                      OUTPUT FUNCTION
                   INQ         -1                      PREPARE FOR DATA INPUT
                   NOP
LOOP               INP         -1                      INPUT TO A FIRST COLUMN (12)
                   ALS         4                       DATA UPPER 1 I BITS ... ZERO
                   STA*        TEMP
                   INP         -1                      INPUT 2nd COLUMN
                   STA*        TEMP+1                  SAVE THE DATA
                   ARS         8                       UPPER 4 BITS IN LOWER 4
                   AND         =XMASKB                 ZERO UPPER 12 BITS
                   ADD*        TEMP                    FIRST WORD PACKED
                   STA*        AA,I
                   LDA*        TEMP+1
                   ALS         8                       LOWER 8 BITS IN UPPER 8 BITS
                   AND         =N$FF00                 ZERO LOWER 8 BITS
                   STA*        TEMP
                   INP         -1                      INPUT 3rd COLUMN
                   STA*        TEMP+1
                   ARS         4                       UPPER INPUT 8 BITS IN LOW 8BITS
                   AND         =X$00FF                 ZERO UPPER 8 BITS
                   ADD*        TEMP                    PACK 2nd WORD
                   STA+        AA+1,I                  PLACE IN BUFFER
                   LDA*        TEMP+1
                   AND         =X MASKB                ZERO UPPER 12 BITS
                   ALS         12                      LOWER 4 BITS IN UPPER 4 BITS
                   STA*        TEMP
                   INP         -1
                   AND         =N$0FFF
                   ADD*        TEMP                    THIRD WORD PACKED
                   STA*        AA+2,I
                   LDA-        I                       GET INDEX
                   INA         3                       UPDATE INDEX
                   STA-        I
                   INA         -54
                   SAZ         1                       IF END OF CARD SKIP
                   JMP*        LOOP                    CONTINUE
                   LDQ         =XCARD
                   RAO         *+3
                   LDA         =N$7FFA
                   SOV         1
                   JMP         SECOND
DONE               SLS
                   END

03FF               CARD        0281                    MASKB  000F  TEMP  0000P
0004P              LOOP        000CP                   DONE  003EP
```

12.6 1742 LINE PRINTER

The 1742 Line Printer (the Holley HR-300 Printer) prints 300 lines per minute, each line being 136 characters. The printer accepts ASCII codes with two characters per 16 bit word. The printer has a holding register capable of accepting an entire line before printing: 136 characters (8 bits) or 68 words (16 bits). The ASCII codes require 7 bits, therefore, the 8th bit is used as a print control bit. This bit on each character is set by the controller when the character is received. As the printer actually prints the character, the 8th bit is cleared. When all print control bits have been zeroed, the printer has completed the print operation and is ready to receive data from the computer. The programmer is not required to send the maximum number of characters to the printer. The printer accepts the characters sent by the program and blanks the remaining positions prior to printing.

The Q Register will address the printer in the following format:



The W portion will equal zero. The equipment number will correspond to the equipment setting of the hardware switch on the controller ($0-$F). The D portion will direct the type of transmissions to and from the A Register.

DATA

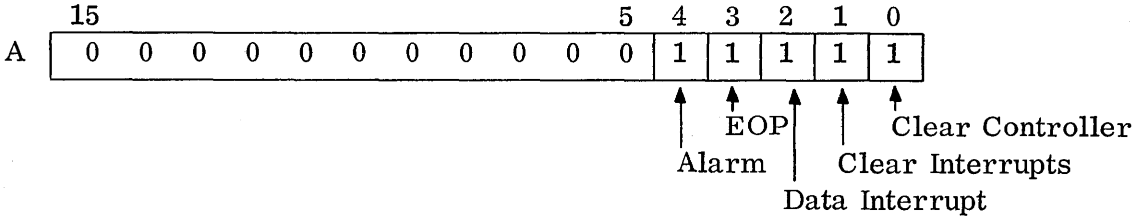
D=00 indicates the transfer of data to the printer's holding register. It will always be issued with an OUT instruction.

```
LDQ        =N$0380        EQUIP 7, DATA
LDA        DATA           PLACE DATA IN A
NOP
OUT        -1             DATA SENT TO PRINTER
```

12.6.1 LP Functions

DIRECTOR FUNCTION 1

D=01 denotes the transfer of Director Function 1, which allows the programmer to CLEAR PRINTER and CLEAR INTERRUPTS. It also provides the medium for selecting as many as three interrupts: DATA, EOP, and ALARM.

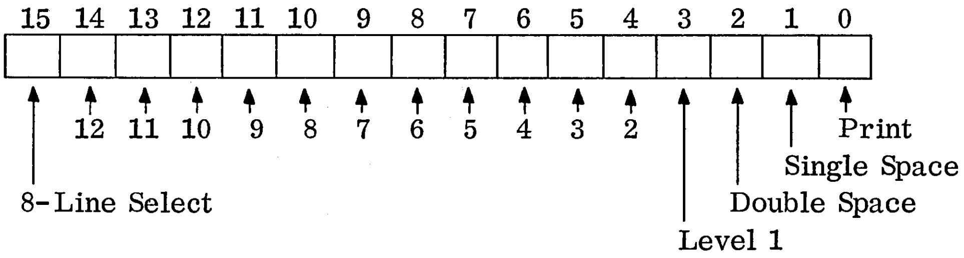


```
LDQ          =N$0381          EQUIP 7, FUNC 1
LDA          FUNC1            PLACE FUNC IN A
OUT          -1               SEND FUNC
```

DIRECTOR FUNCTION 2

D=11 accompanied by an OUT instruction sends Director Function 2 to the printer from the A Register.

```
LDQ          =N$0383          EQUIP, FUNC 2
LDA          FUNC2            PRE SET A TO FUNC
NOP
OUT          -1               SEND FUNC FROM A
```



Print (Bit 0=1) ——► Commands the printer operation to begin. Once the entire line has been printed the operation is complete. The print command must be issued for each line printed.

Single Space
(Bit 1 = 1) ——► Advances the page by one line.

Double Space
(Bit 2 = 1) ——► Advances the page by two lines.

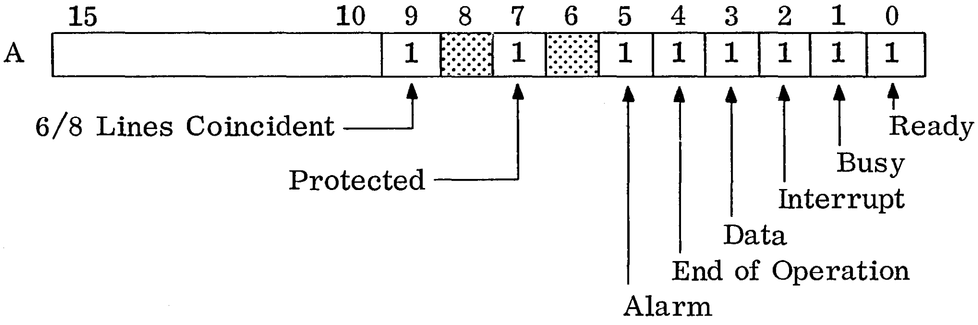
Format Level
(Bit 3 - Bit 14) ——► This command causes paper motion. The paper is moved to the next hole punched in the specified level. The tape levels are used for formatting documents, as the levels may be used instead of the spacing commands. Level 1 usually specifies top of page while level 12 usually specifies bottom of page.

8-Line Select
(Bit 15 = 1) ——► This bit changes the logic to allow for 8 lines per inch rather than 6 lines per inch. Once 8 lines per inch has been selected it remains in effect until a MASTER CLEAR, CLEAR CONTROLLER, or bit 15 is issued as a zero.

12.6.2 LP Status

D=01 with an INP instruction requests status.

```
LDQ         =N$0381         EQUIP #7, STATUS
NOP
INP         -1              BRING STATUS TO A
```



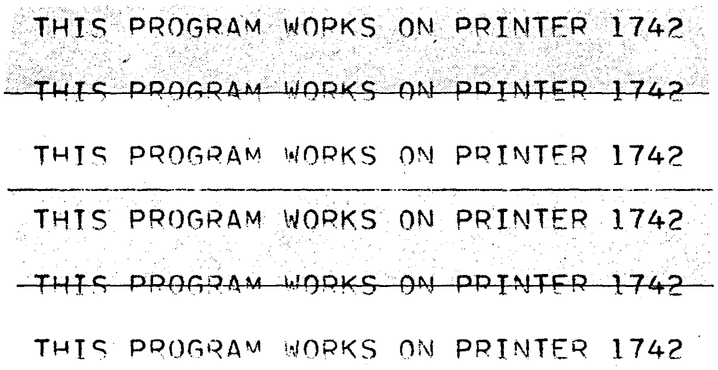
The status of the Line Printer notifies the program of the READY and BUSY states of the printer by setting bits 0 and 1, respectively. Bit 2 indicates the existence of an interrupt, while bits 3, 4, and 5, designate which interrupt was generated. Bit 3 is the bit indication that the printer is prepared to receive data from the computer. Bit 4 notifies the program that an operation is complete, such as a print or top of form command. Bit 5 indicates an ALARM condition has occurred, such as out of paper, paper tear, fuse alarm, open interlock, or an illegal character. Bit 7 corresponds with the Protect Switch on the printer. If the switch is on, bit 7 will be set and only protected programs will be allowed to use the printer. Bit 9, when set, acknowledges that a change from 6 to 8 lines per inch may be effectively made.

12.6.3 Programming the Printer

Programming the printer simply requires advancing the page to the desired print level. Once the page is positioned, data is sent to the printers holding register. Once the data has been sent to the printer, the print command is issued. When the printer completes the print cycle, the program advances the page by level selection, or spacing and outputs the next line. The programmer selects the top of page level (usually Level 1) once a page is complete.

12.6.4 Example, 1742 Line Printer

```
0001                              NAM        PRINT
0002                              ENT        PRINT
0003 P0000 E000     PRINT         LDQ        =N$0781      EQUIP 15, FUNC 1
     P0001 0781
0004 P0002 0A01                   ENA        1            CLR CONTROLLER
0005 P0003 03FE                   OUT        -1
0006 P0004 E000                   LDQ        =N$0783      EQUIP 15, FUNC 2
     P0005 0783
0007 P0006 0A08                   ENA        8            LEVEL 1 TOP OF FORM
0008 P0007 03FE                   OUT        -1
0009 P0008 C000                   LDA        =X$7FFF-5    LOOP FOR OVERFLOW
     P0009 7FFA
0010 P000A 6826                   STA*       LINECT       6 LINES PER PAGE
0011 P000B 0844     CNT           CLR        A
0012 P000C 60FF                   STA-       I            ZERO INDEX
0013 P000D 0DFC                   INQ       -3            PRINTER FOR DATA
0014 P000E C911     OUT           LDA*       DATA,I       2 ASCII CHARACTERS
0015 P000F 0800                   NOP
0016 P0010 03FE                   OUT        -1
0017 P0011 C0FF                   LDA-       I            CHECK WORD LOOP
0018 P0012 09FE                   INA        -$11         17 CHARACTER PER LINE
0019 P0013 0102                   SAZ        NEXTL        SKIP IF COMPLETE
0020 P0014 D0FF                   RAO-       I            NOT COMP, UPDATE INDEX
0021 P0015 18F8                   JMP*       OUT          CONTINUE LINE OUTPUT
0022 P0016 0D03     NEXTL         INQ        3            LINE COMPLETE, FUNCTION
0023 P0017 0A01                   ENA        1            PRINT COMMAND
0024 P0018 03FE                   OUT        -1
0025 P0019 0A04                   ENA        4            DOUBLE SPACE
0026 P001A 03FE                   OUT        -1
0027 P001B D815                   RAO*       LINECT       UPDATE LINE COUNT
0028 P001C 01A1                   SOV        CMP          SKIP IF PAGE COMPLETE
0029 P001D 18FD                   JMP*       CNT          CONTINUE IF PAGE NOT COMPLETE
0030 P001E 0000     CMP          SLS
0031 P001F 5448     DATA          ALF        *,THIS PROGRAM WORKS ON PRINTER 1742*
     P0020 4953
     P0021 2050
     P0022 524F
     P0023 4752
     P0024 414D
     P0025 2057
     P0026 4F52
     P0027 4B53
     P0028 204F
     P0029 4E20
     P002A 5052
     P002B 494E
     P002C 5445
     P002D 5220
     P002E 3137
     P002F 3432
0032 P0030 0001                   BSS        LINECT(1)
0033                              END        PRINT
```



THIS PROGRAM WOPKS ON PRINTFR 1742

~~THIS PROGRAM WORKS ON PRINTER 1742~~

THIS PROGRAM WORKS ON PRINTER 1742

THIS PROGRAM WORKS ON PRINTER 1742

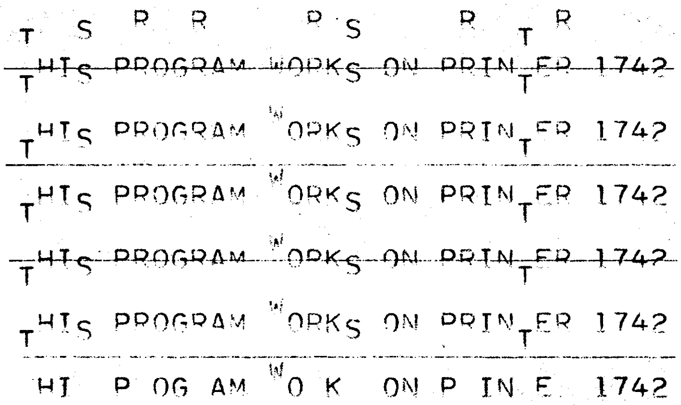
~~THIS PROGRAM WORKS ON PRINTER 1742~~

THIS PROGRAM WORKS ON PRINTER 1742

Note what happens in the following program when the functions for print command and double space are issued simultaneously! (Line 23, ENA 5.)

12.6.4

```
0001                            NAM       PRINT
0002                            ENT       PRINT
0003 P0000 E000    PRINT        LDQ       =N$0781
     P0001 0781
0004 P0002 0A01                 ENA       1          CLR CONT
0005 P0003 03FE                 OUT       -1
0006 P0004 E000                 LDQ       =N$0783
     P0005 0783
0007 P0006 0A08                 ENA       8          TOP OF FORM
0008 P0007 03FE                 OUT       -1
0009 P0008 C000                 LDA       =X$7FFF-5
     P0009 7FFA
0010 P000A 6824                 STA*      LINECT
0011 P000B 0344    CNT          CLR       A
0012 P000C 60FF                 STA-      I
0013 P000D 0DFC                 INQ      -3
0014 P000E C90F    OUT          LDA*      DATA,I
0015 P000F 0B00                 NOP
0016 P0010 03FE                 OUT       -1
0017 P0011 C0FF                 LDA-      I
0018 P0012 09EE                 INA       -$11
0019 P0013 0102                 SAZ       NEXTL
0020 P0014 D0FF                 RAO-      I
0021 P0015 18F8                 JMP*      OUT
0022 P0016 0D03    NEXTL        INQ       3
0023 P0017 0A05                 ENA       5
0024 P0018 03FE                 OUT       -1
0025 P0019 D815                 RAO*      LINECT
0026 P001A 01A1                 SOV       CMP
0027 P001B 18EF                 JMP*      CNT
0028 P001C 0000    CMP          SLS
0029 P001D 5448    DATA         ALF       *,THIS PROGRAM WORKS ON PRINTER 1742*
     P001E 4953
     P001F 2050
     P0020 524F
     P0021 4752
     P0022 414D
     P0023 2057
     P0024 4F52
     P0025 4B53
     P0026 204F
     P0027 4E20
     P0028 5052
     P0029 494E
     P002A 5445
     P002B 5220
     P002C 3137
     P002D 3432
0030 P002E 0001                 BSS       LINECT(1)
0031                            END       PRINT
```



```
 T   S   R   R      R S      R   T R
 HIS PROGRAM WORKS ON PRIN ER 1742
T                             T
         W
 HIS PROGRAM  ORKS ON PRIN ER 1742
T                         T
         W
 HIS PROGRAM  ORKS ON PRIN ER 1742
T                         T
         W
 HIS PROGRAM  ORKS ON PRIN ER 1742
T                         T
         W
 HIS PROGRAM  ORKS ON PRIN ER 1742
T                         T
         W
 HI  P OG AM  O K  ON P IN E  1742
```

Digigraphics - September 1969

12.7 1738/853 DISK

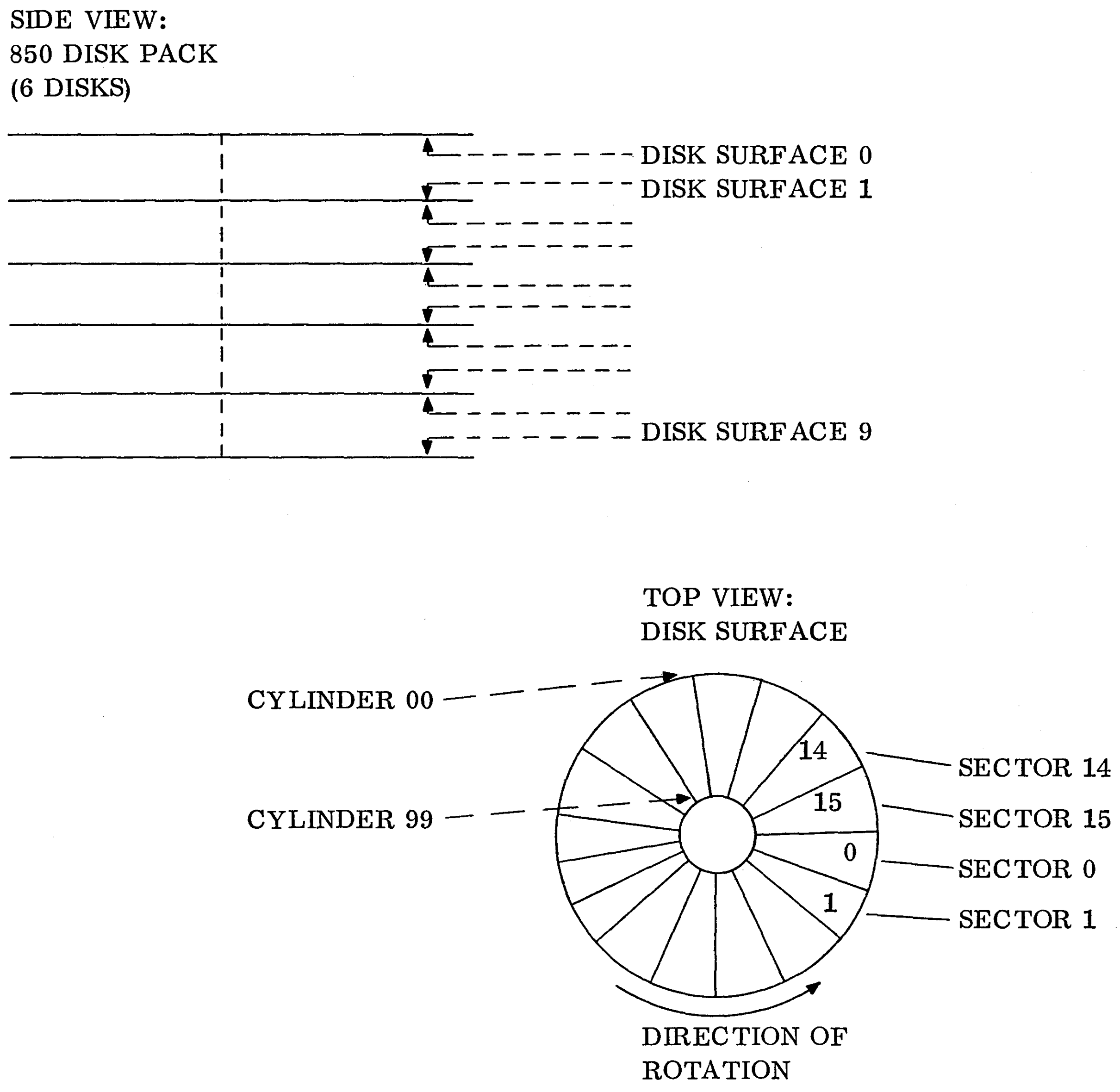
The disk is a buffered peripheral device attached to the 1705 Direct Access Bus. All data will be buffered in and out of memory via the 1705 in 16-bit words. The functions will be sent from the A Register and status will be received in the A Register, necessitating the connection to the A/Q channel. The disk transfers 16-bit data words in 12.8 microseconds. Access time for positioning the head is 165 milliseconds maximum. Cylinder-to-cylinder positioning time is 30 milliseconds. The disk has a maximum latency time of 25 milliseconds.

Ninety-six 16-bit words may be stored on one sector with 1,536 words on a track and 15,360 words to a cylinder. The 853 disk pack allows a total of 1,536,000 words, while the 854 disk pack has a capacity of 3,118,080 words.

The data format may be summarized in the following:

16	Bit data words
96	Words to a sector
16	Sectors to a track
10	Tracks to a cylinder
100	Cylinders to an 853 file
203	Cylinders to an 854 file

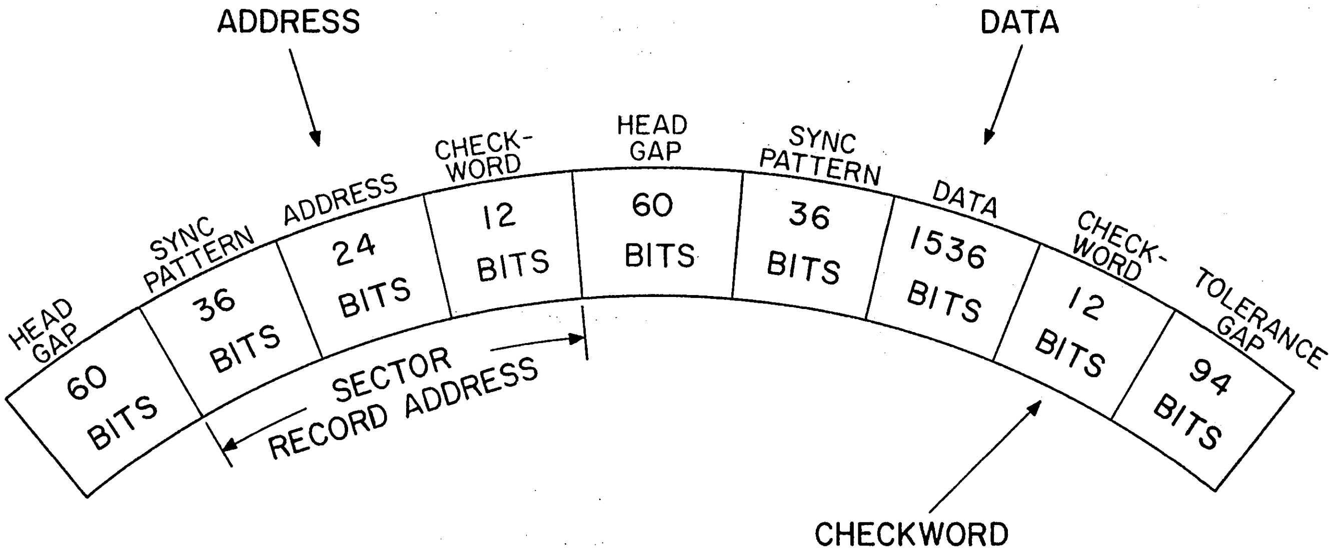
The three interfaces for communications with the 1738 controller are the A/Q channel, DAC (1705) and the CONTROLLER/FILE. The A/Q channel is the interface between the controller and the programmer. It is via the A/Q channel that the programmer may status the disk and issue functions. The DAC is the interface between the controller and the computer's memory. It is via this channel that the data is transferred. The DAC also provides the 1738 access to the LWA+1 of the programmer's buffer area. CONTROLLER/FILE INTERFACE is used for communications between the controller and the disk. It is through this interface that the controller informs the disk of the Sector Record Address selected by the programmer. The SEEK operation which positions the read/write heads to the SECTOR RECORD ADDRESS is generated by the controller once the controller receives the desired address from the A/Q interface. The controller will issue a SEEK FORWARD or SEEK REVERSE command depending upon the current position of the read/write heads. It does not return to a set address prior to positioning on a new address.



853 contains 100 cylinders; 854 contains 203 cylinders.

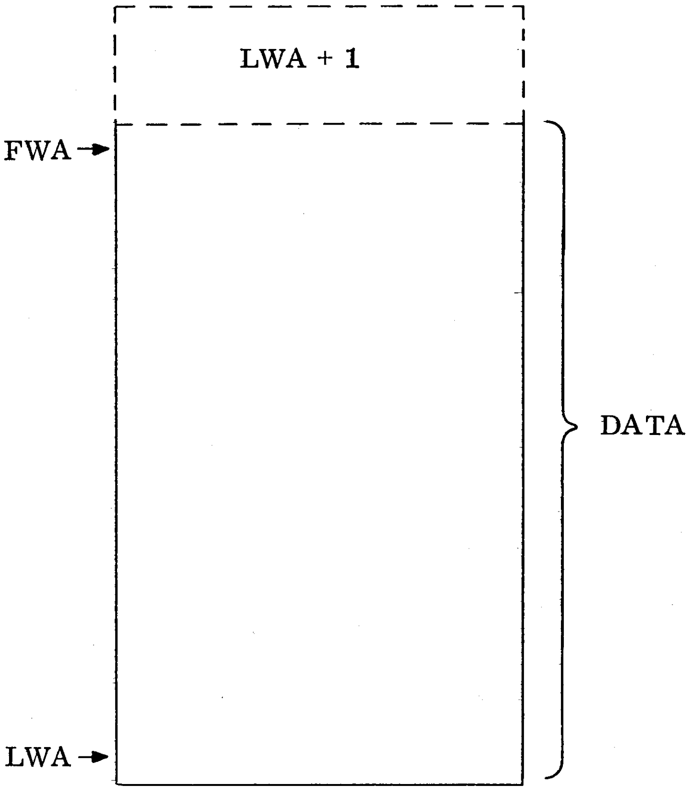
Figure 29. Disk

Figure 30. Sector Format on Disk



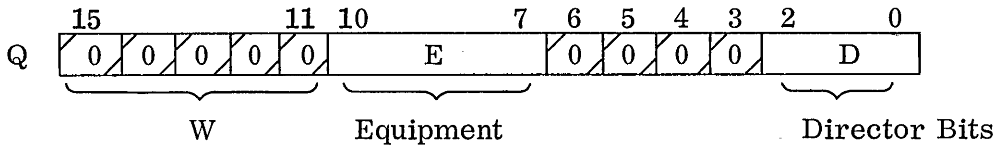
Each sector on the disk contains the above information. Note that the 96 16-bit words of data (1536 data bits) are in addition to the other check bits in the sector.

Figure 31. Data Buffer for Disk



FWA-1 must contain LWA+1 of buffer.

The Q Register will contain the address of the disk.



W field is zero and E field contains equipment number (set on controller).

The setting of the director bits will define the desired operation to the controller. The contents of the A Register will vary according to the director bits.

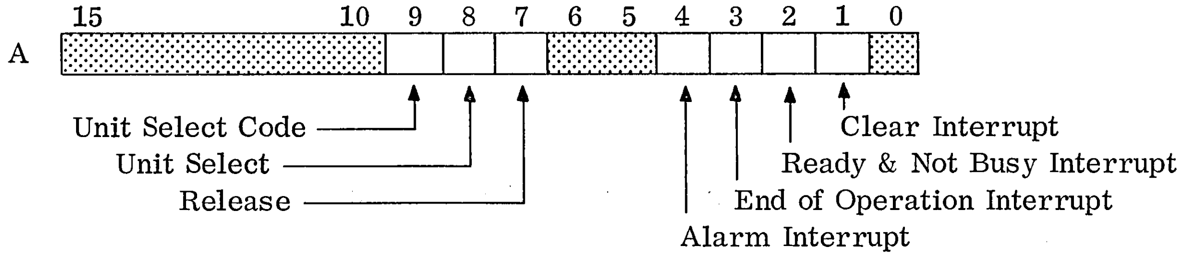
12.7.1 Disk Functions

DISK FUNCTION CODES

Value Set in Q (Bits 02 - 00)	Output from A	Input to A
001	Director Function	Director Status
010	Load Address	Address Register Status
011	Write	
100	Read	
101	Compare	
110	Checkword Check	
111	Write Address	

12.7.1.1 Director Bits 001 - Director Functions

This setting with an OUT instruction prepares the controller for director functions which are found in the A Register. The functions for the disk may all be sent at the same time.



The CLEAR INTERRUPT function will clear all selected interrupts allowing the programmer to select the interrupts he desires. Three interrupts may be selected: NEXT READY AND NOT BUSY STATUS, END OF OPERATION, and ALARM. The NEXT READY AND NOT BUSY interrupt occurs when the 1738 becomes not busy, but still maintains its ready status. This interrupt can be used during an overlap seek. The overlap seek is used when two disks are connected to one controller. The programmer may issue a sector record address for one disk and then issue a sector record address for the other. The controller will generate a NEXT READY AND NOT BUSY interrupt, if selected by the programmer, when one of the disks reaches the requested address.

The END OF OPERATION interrupt allows the controller to inform the 1700 when it has completed an operation such as a data transfer. The ALARM INTERRUPT will notify the 1700 that an alarm condition has arisen. There are eight possible alarm conditions; not ready, checkword error, lost data, seek error, address error, defective track, storage parity error, and protect fault.

The RELEASE function allows an unprotected program to use the disk even though the protect switch on the disk is still set. A protected program must issue the release function. The next time a protected program accesses the disk, the disk will become protected and must again be released before the disk will become accessible to an unprotected program.

The UNIT SELECT and UNIT SELECT CODE will always be zero unless two disks are connected to the 1738. Bit 8 is the UNIT SELECT bit which informs the controller that the program will select unit 0 or unit 1. Bit 9 indicates which unit bit 8 wishes to select. If bit 9 is a 0, unit 0 is selected; if it is a 1, unit 1 is selected. The controller ignores bit 9 unless bit 8 is set.

12.7.1.2 Director Bits 010 - Sector Record Address

This director code in the Q Register with an OUT instruction will send the SECTOR RECORD ADDRESS from the A Register to the controller. Once the controller receives the address, it initiates the seek operation. The SECTOR RECORD ADDRESS will be in the following format:

	15 8	7 4	3 0
A	CYLINDER	HEAD	SECTOR

12.7.1.3 Director Bits 011 - WRITE

The WRITE function code requests the controller to prepare to read data from memory and write it on the disk. Prior to this function, the programmer must send the SECTOR RECORD ADDRESS to the controller.

The controller expects to find the first word address minus 1 (FWA-1) of the buffer area in the A Register when the write function is received. The controller goes into memory via the DAC to the FWA-1 at which location he extracts the last word address plus 1 (LWA+1). The controller keeps the LWA+1 and updates the FWA-1 until the two are equal at which point the write operation is complete.

Prior to issuing the WRITE function, the SECTOR RECORD ADDRESS must be sent to the controller and the LWA+1 of the buffer area must be at the FWA-1.

12.7.1.4 Director Bits 100 - READ

The READ function code follows the same programming procedure as the WRITE function. The difference being the disk reads data into memory rather than writing data on the disk. An unprotected program may READ from a protected disk without generating a protect FAULT, however, if an unprotected program attempts to write on a protected disk, a protect fault will occur. An alarm interrupt will be generated if previously selected.

12.7.1.5 Director Bits 101 - COMPARE

The COMPARE function code follows the same programming procedure as the READ and WRITE function codes. The COMPARE function causes the controller to read data from the computer's memory and compare it with the data stored on the disk. If at any time during the compare, one word does not compare, the NO COMPARE status bit will be set. This function provides an extra check on the validity of the data transferred.

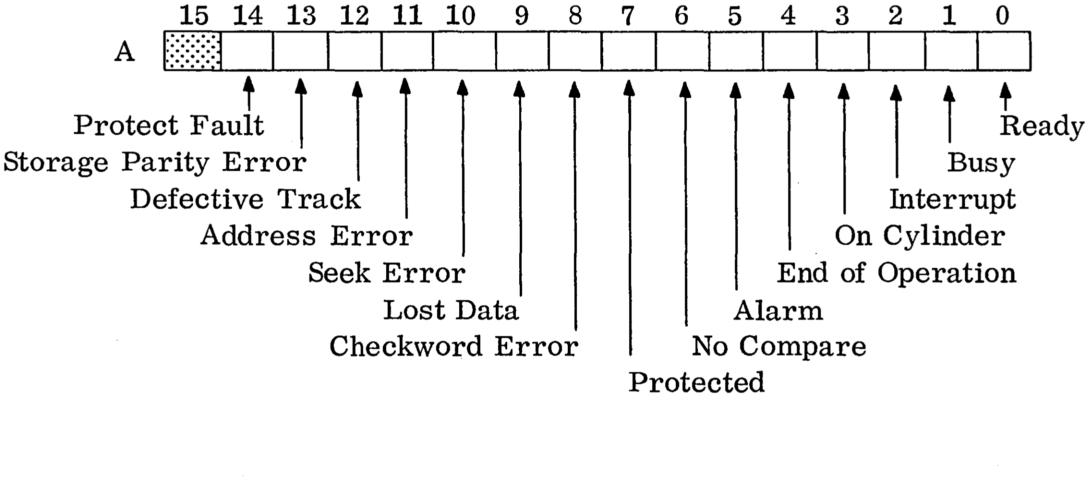
12.7.1.6 Other Director Functions

The remaining director functions (CHECKWORD CHECK and WRITE ADDRESS) are used by the customer engineers for maintenance work.

12.7.2 Disk Status

12.7.2.1 Director Status

D = 001 accompanied by INP instruction, will request the 1738 to send status to the A Register.



The READY status indicates that the unit is available. The BUSY bit indicates that the controller and/or the drive unit is presently involved in the performance of an operation. This bit is set with the acceptance of a LOAD ADDRESS, WRITE, READ, COMPARE, CHECKWORD CHECK, or WRITE ADDRESS function. At the completion of the function which set the BUSY status, the status will be cleared and the disk will become NOT BUSY. Once the disk is NOT BUSY, a new function may be issued.

The INTERRUPT bit acknowledges that an interrupt has occurred. Further examination of A will determine which of the three selected interrupts was generated; bit 4 (EOP) and bit 5 (ALARM). If neither bit 4 nor bit 5 is set, the programmer should check bits 0 and 1 for READY and NOT BUSY. If the alarm bit is set, the programmer must evaluate A further to determine which of the eight alarm conditions caused the interrupt.

The ON CYLINDER status, bit 3, is set when the READ/WRITE heads have reached the SECTOR RECORD ADDRESS initially sent to the controller via the A/Q channel.

12.7.2.2 Address Register Status

The D=010 Q setting accompanied by an INP instruction will direct the controller to return the current sector record address of the disk to the A Register, the location at which the READ/WRITE heads are currently positioned. It will be in the same format as described in the Address Function.

12.7.3 Disk Sample Programs

```
0001                          NAM   DISK
0002                          ENT   DISK
0003 P0000 C000    DISK       LDA   =XLWPO          LWA+1 IN 'A'
     P0001 0092 P
0004 P0002 6800               STA   FWMO            LWA+1 AT FWA-1
     P0003 002E
0005 P0004 D800               RAO   FLAG            UPDATE FLAG FOR LOOP
     P0005 0011
0006 P0006 E000               LDQ   =N$0181         EQUIP 3, FUNCTION
     P0007 0181
0007 P0008 0402               ENA   $0002           CLR INT
0008 P0009 03FE               OUT   -1
0009 P000A E000    ZAP        LDQ   =N$0182         EQUIP 3, LOAD ADDRESS
     P000B 0182
0010 P000C 0A10               ENA   $0010           CYL 0, HEAD 1, SECTOR 0
0011 P000D 03FE               OUT   -1
0012 P000E E000               LDQ   =N$0181         EQUIP 3, STATUS
     P000F 0181
0013 P0010 0B00               NOP
0014 P0011 02FE    STAT       INP   -1              STATUS IN A
0015 P0012 0FCC               ALS   12              ON CYL
0016 P0013 0131               SAM   1               SKIP WHEN ON CYL
0017 P0014 18FC               JMP*  STAT            WAIT
0018 P0015 1C00               NUM   $1C00           JMP (0)
0019 P0016 0000    FLAG       0     0
0020 P0017 0019 P             ADC   WRITE           GO WRITE
0021 P0018 002A P             ADC   READ            GO READ
0022 P0019 C000    WRITE      LDA   =XFWMO          FWA-1 IN A
     P001A 0031 P
0023 P001B E000               LDQ   =N$0183         EQUIP 3, WRITE
     P001C 0183
0024 P001D 0B00               NOP
0025 P001E 03FE               OUT   -1
0026 P001F E000    XX         LDQ   =N$0181         EQUIP 3, STATUS
     P0020 0181
0027 P0021 0B00               NOP
0028 P0022 02FE    LOOP       INP   -1              STATUS IN 'A'
0029 P0023 0FCB               ALS   11              CH FOR END OF OP
0030 P0024 0131               SAM   1               SKIP WHEN COMP
0031 P0025 18FC               JMP*  LOOP            WAIT
0032 P0026 0000               SLS   0               STOP TO ZERO BUF FROM CONSOLE
0033 P0027 D800               RAO   FLAG            ADD 1 FLAG READ
     P0028 FFED
0034 P0029 18E0               JMP*  ZAP             CONTINUE, READ
0035 P002A C000    READ       LDA   =XFWMO          FWA-1 IN A
     P002B 0031 P
0036 P002C E000               LDQ   =N$0184         EQUIP 3, READ
     P002D 0184
0037 P002E 0B00               NOP
0038 P002F 03FE               OUT   -1
0039 P0030 18EE               JMP*  XX              GO STATUS FOR END OF OP
0040 P0031 0001               BZS   FWMO(1)
0041 P0032 0060               BZS   FWA(96)
0042 P0092 0001               BZS   LWPO(1)
0043                          END   DISK
```

```
0001                          NAM      DISK
0002 P0000 E000    START      LDQ      =N$0181      EQUIP 3, FUNCTION
     P0001 0181
0003 P0002 0844               CLR      A            INITIALIZE INDEX
0004 P0003 60FF               STA-     I
0005 P0004 C000               LDA      =N$0002      CLEAR INTERRUPTS
     P0005 0002
0006 P0006 0B00               NOP
0007 P0007 03FE               OUT      -1
0008 P0008 E000               LDQ      =N$0182      SECTOR RECORD ADDRESS
     P0009 0182
0009 P000A C000               LDA      =N$460       CYL 4,HEAD 6, SECTOR 0
     P000B 0460
0010 P000C 0B00               NOP
0011 P000D 03FE               OUT      -1
0012 P000E E000               LDQ      =N$0181      EQUIP 3, STATUS
     P000F 0181
0013 P0010 0B00               NOP
0014 P0011 02FE    TEST1      INP      -1           STATUS IN A
0015 P0012 0F43               ARS      3            CHECK ON CYL
0016 P0013 A000               AND     =XMSK
     P0014 0001
0017 P0015 0111               SAN      WRITE        SKIP IF ON CYL
0018 P0016 18FA               JMP*     TEST1        WAIT UNIT ON CYL
0019 P0017 C000    WRITE      LDA      =XBUF+96     LWA+1
     P0018 00B7 P
0020 P0019 6800               STA      BUF-1        STORE AT FWA-1
     P001A 003C
0021 P001B C000               LDA      =XBUF-1      LOAD 'A' FWA-1
     P001C 0056 P
0022 P001D E000               LDQ      =N$0183      EQUIP 3, WRITE
     P001E 0183
0023 P001F 0B00               NOP
0024 P0020 03FE               OUT      -1
0025 P0021 E000               LDQ      =N$0181      EQUIP 3, STATUS
     P0022 0181
0026 P0023 0B00    TEST2      NOP
0027 P0024 02FE               INP      -1           STATUS IN A
0028 P0025 0F44               ARS      4            CK END OF OP
0029 P0026 A000               AND     =XMSK
     P0027 0001
0030 P0028 0111               SAN      ZERO         SKIP WHEN COMPLETE
0031 P0029 18F9               JMP*     TEST2        CONTINUE WAITING
0032 P002A 0844    ZERO       CLR      A            ZERO BUFFER AREA
0033 P002B 6900               STA      BUF,I
     P002C 002B
0034 P002D D0FF               RAO-     I            UPDATE I
0035 P002E C000               LDA      =XCOUNT
     P002F 0060
0036 P0030 90FF               SUB-     I
0037 P0031 0132               SAM      READ         IF COM SKIP
0038 P0032 1800               JMP      ZERO         CONTINUE ZERO UNTIL COMPLETE
     P0033 FFF6
0039 P0034 0000    READ       SLS                   STOP TO CK BUFFER
0040 P0035 E000               LDQ      =N$0182      EQUIP 3, LOAD ADDR
     P0036 0182
```

```
0041 P0037 C000           LDA     =N$460    CYL 4, HEAD 6, SEC 0
     P0038 0460
0042 P0039 0B00           NOP
0043 P003A 03FE           OUT     -1
0044 P003B E000   TEST3   LDQ     =N$0181   EQUIP 3, STATUS
     P003C 0181
0045 P003D 0B00           NOP
0046 P003E 02FE           INP     -1        STATUS IN  'A'
0047 P003F 0F43           ARS     3         CK ON CYL
0048 P0040 A000           AND     =XMSK
     P0041 0001
0049 P0042 0111           SAN     READ1     SKIP WHEN ON CYL
0050 P0043 18F7           JMP*    TEST3     WAIT
0051 P0044 C000   READ1   LDA     =XBUF-1   FWA-1 IN 'A'
     P0045 0056 P
0052 P0046 E000           LDQ     =N$0184   EQUIP 3, READ
     P0047 0184
0053 P0048 0B00           NOP
0054 P0049 03FE           OUT     -1        OPERATION INITIATED
0055 P004A E000   TEST4   LDQ     =N$0181   EQUIP 3, STATUS
     P004B 0181
0056 P004C 0B00           NOP
0057 P004D 02FE           INP     -1        STATUS IN A
0058 P004E 0F44           ARS     4         CK END OP
0059 P004F A000           AND    =XMSK
     P0050 0001
0060 P0051 0111           SAN     EXIT      SKIP WHEN COMP
0061 P0052 18F7           JMP*    TEST4     WAIT
0062 P0053 0000   EXIT    SLS               STOP
0063 P0054 1800           JMP     START     BEGIN AGAIN
     P0055 FFAA
0064       0001           EQU     MSK(1)
0065       0060           EQU     COUNT(96)
0066 P0056 0001           BSS     (1)
0067 P0057 0060           BSS     BUF(96)
0068                      END
```

12.7.3.1 Address Tag Program

Write Addresses:

Every new disk pack has to have addresses written on it before it can be used for data storage. Each sector must have an address tag. The hardware address tag switch and the write address function code are for writing the addresses. The following program could be used to write tags.

```
                 NAM          WRITE ADDRESS TAGS
*                                                                 *
*           TURN ADDR TAG SWITCH ON * DISK IS EQUIP 3             *
*           FOR 854 CHANGE CYL EQU TO $CB                         *
*                                                                 *
                 EQU          EQUIP($0182),CYL($64)
                 ENT          TAGS
TAGS             0            0
                 LDA          =N$0102         *SEL UNIT 0, CLR INT
                 LDQ          =XEQUIP-1       *DIR FUNC
                 NOP
                 OUT          -1
                 INQ          1               *SEEK FUNC 010
                 ENA          0               *ADDRESS 0000
LOOP             OUT          -1
                 INQ          5               *WRITE TAG FUNC 111
                 NOP
                 OUT          -1
                 INQ          -6              *DUMMY DIR FUNC 001
                 ENA          0               *LAZY MAN'S BUSY CK
                 OUT          -1              *FALLS THRU WHEN BUSY
                 INQ          1               *LOAD ADDR FUNC 010
                 NOP
                 INP          -1              *NEXT ADDR IN A
                 EOR          =XCYL           *FINISHED?
                 SAZ          EXIT
                 NOP
                 INP          -1              *GET NEXT ADDR BACK
                 JMP*         LOOP
EXIT             SLS                          *STOP
                 JMP*         TAGS+1          *GO DO IT AGAIN
                 END
```

12.7.3.1

Usually the program to write tags is keyed in from the console rather than run in assembly language. Therefore, it would be desirable to shorten the program. Error checks can be eliminated if the hardware is functioning properly. The following code is used by the customer engineers:

```
E000
0182
02FE
0D05
03FE
0DF9
0A00
03FE
18F7
```

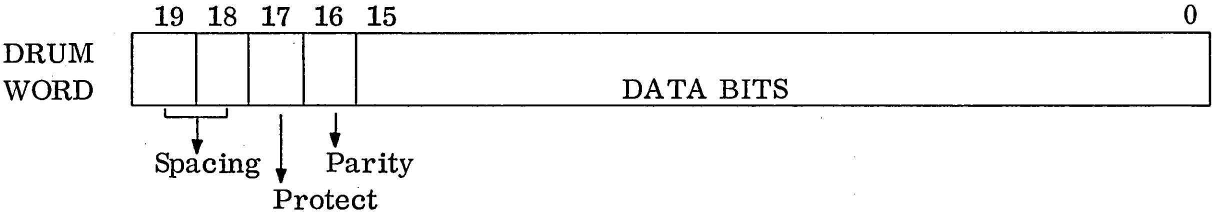
Decode the program and see what it does. A master clear sets the disk at address 0000 to begin. The program will stop on alarm when it is finished and is attempting to write an address beyond the last cylinder (on an 853 or 854).

12.7.4 Problem

Write a program to write zeroes on the entire disk pack after the new address tags have been written. Include error checks.

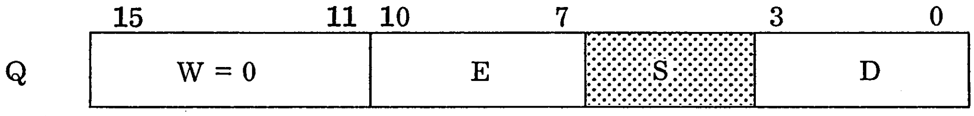
12.8 1751 DRUM CONTROLLER

The 1751 Drum Controller interfaces with drums ranging in size from $65,536_{10}$ words to $8,388,508_{10}$ words. The drum word size is 20 bits composed of 16 data bits, 1 parity bit (odd), 1 protect bit and 2 spacing bits.



The transfer rate for one word is 8 microseconds. All data transfers to and from the 1700 are via the DAC. The access time for the drum is 8 milliseconds, with a maximum of 16 milliseconds.

The Q Register will be in the following format when addressing the 1751.



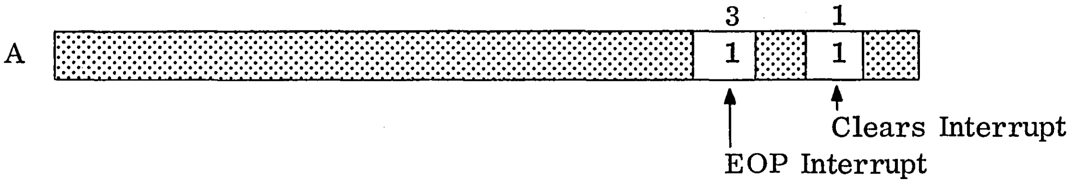
The D portion of Q determines the type of information being sent or received in the A Register. The INP and OUT instructions, accompanied by the Q setting, control the information flow to A (INP) and from A (OUT).

Figure 32. Interim Drum Interface Codes

1700 I/O	Q03	Q02	Q01	Q00	DESCRIPTION
Write	x	x	x	1	Director function
					A_0 = Not used
					A_1 = 1 Clear Interrupt
					A_2 = Not used
					A_3 = 1 End of Operation Interrupt Request
Write	0	a	b	0	Initiate Operation
	0	0	0	0	ab = 00 Write Data From Core
	0	0	1	0	ab = 01 Write Zeros
	0	1	0	0	ab = 10 Read Data to Core
	0	1	1	0	ab = 11 Check Parity on Drum
Write	1	a	b	0	Load Address Register
	1	0	0	0	ab = 00 Track
	1	0	1	0	ab = 01 Initial Sector
	1	1	0	0	ab = 10 Initial Core
	1	1	1	0	ab = 11 Final Core
Read	x	x	0	1	Director Status I
					A_0 = 1 Ready
					A_1 = 1 Busy
					A_2 = 1 Interrupt
					A_3 = Not used
					A_4 = 1 End of Operation
					A_5 = 1 Not used
					A_6 = 1 Lost Data
					A_7 = 1 Protected
					A_8 = 1 Parity Error
					A_9 = 1 Not used
					A_{10} = 1 Guarded Address
					A_{11} = 1 Timing Track Error
	x	x	1	1	Director Status II
					A_0 - A_{11} Sector Address

12.8.1 Drum Functions

When the D portion equals 0001_2 accompanied by an OUT instruction, the A Register must be preset. The setting of A determines the function or functions to be sent to the 1751.



The Clear Interrupt bit clears the interrupt. The EOP INT (End of Operation Interrupt) takes precedence over the CLR INT. When the EOP bit is set, the 1751 will generate an interrupt when it has completed an operation. The remaining bits in A are not used, therefore, they should be set to zeros.

When programming the drum the programmer first clears interrupts. If writing in interrupt mode, he should also select the EOP interrupt.

```
LDQ        =N$0101        EQUIP 2, DRUM FUNC
ENA        $000A          CLR INT, SEL EOP
OUT        -1             OUTPUT FUNCTION
```

Once the interrupts have been cleared and reselected, the programmer must tell the controller the first word address (FWA) of his buffer area in core memory, as well as the last word address (LWA) of the core memory buffer. This is accomplished by two D settings: D = 1100_2 denotes FWA, D = 1110_2 indicates the last word address. These settings are accompanied by an OUT instruction with the address preset in the A Register.

```
LDQ        =N$010C        EQUIP 2, FWA
LDA        =XFWA          A = FWA
NOP
OUT        -1             OUTPUT ADDRESS

LDQ        =N$010E        EQUIP 2, LWA
LDA        =XLWA          A = LWA
NOP
OUT        -1             OUTPUT ADDRESS
```

The controller then knows the area and length of the computer buffer area. Once the controller knows the memory limits, the programmer must give the drum area by sending the beginning track address and sector address. Both are sent from the lower 12 bits of the A Register. The D portion of the Q Register distinguishes between sector and track address: D = 1000_2 indicates TRACK, D = 1010_2 specifies sector. The programmer may select any one of 4096_{10} tracks. (Note: Not all systems have the maximum number of tracks, therefore, check your configuration.) The programmer may select one of 2048_{10} sectors. (A sector is the drum address of a word within a track.)

```
LDQ       =N$0108      EQUIP 2, TRACK ADDR
ENA       $0004        TRACK 4
OUT       -1           OUTPUT TRACK NUMBER

LDQ       =N$010A      EQUIP 2, SECTOR ADDR
ENA       0            SECTOR 0
OUT       -1           OUTPUT SECTOR NUMBER
```

The controller now knows the core memory and drum memory to be used for an operation. The programmer must now specify one of four operations. The operations are also indicated by the D setting of the Q Register in conjunction with an OUT instruction. The four operations are as follows:

D = 0000_2 initiates a write operation. This write operation instructs the 1751 to write data on the drum from core memory.

D = 0010_2 instructs the 1751 to write zeros on the designated drum area. No data is transferred from memory.

D = 0100_2 initiates a read operation. The read operation transfers data from the drum to core memory.

D = 0110_2 initiates a check operation. The check operation causes the designated drum area to be read and checked for parity errors without any transfer of data into core memory.

```
LDQ       =N$0100      EQUIP 2, WRITE
NOP
OUT       -1           INITIATE WRITE

LDQ       =N$0102      EQUIP 2, WRITE ZEROS
NOP
OUT       -1           INITIATE ZERO WRITE

LDQ       =N$0104      EQUIP 2, READ
NOP
OUT       -1           INITIATE READ

LDQ       =N$0106      EQUIP 2, CHECK
NOP
OUT       -1           INITIATE CHECK
```

12.8.1

The drum at this point will be in the process of performing an operation. If the END OF OPERATION interrupt were selected, the 1751 will generate an interrupt when the operation is completed.

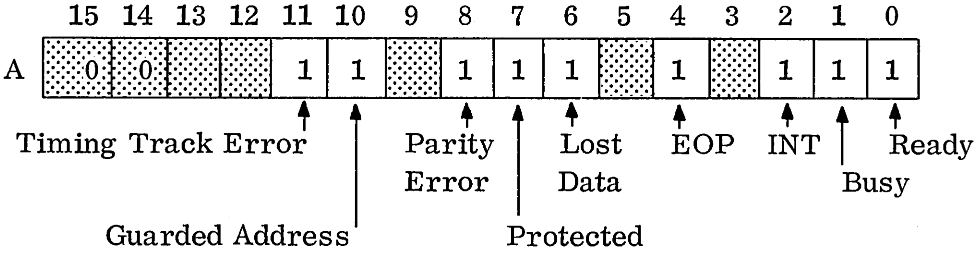
12.8.2 Drum Status

The programmer may take status while the operation is being performed in order to monitor the progress of the operation. He may also take status again at the end of the operation to verify an error free operation.

12.8.2.1 Director Status I

Status may be requested by a D setting of 0001_2 accompanied by an INP instruction. The status will be brought into the A Register.

```
LDQ        =N$0101        EQUIP 2, DIRECTOR STATUS
NOP
INP        -1             BRINGS STATUS INTO A
```



	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
A	0	0	▨	▨	1	1	▨	1	1	1	▨	1	▨	1	1	1

A 1 in the corresponding bit indicates that the stated status exists. For example, a 1 in bit 11 indicates a timing track error.

Timing Track Error

Bit 11 is an error in the timing track which insinuates a hardware problem. The programmer should attempt the operation three or four times before accepting the status as a hardware failure.

Guarded Address

Bit 10 indicates that a core to drum transfer was attempted to a track with an address lower than the one set on the track protect switch.

Parity Error

Bit 8 indicates that the parity was not odd, i.e., it did not have an odd number of one bits in the word.

Protect

Bit 7 indicates that the protect switch on the drum has been set.

Lost Data

Bit 6 indicates that data was not transferred from the controller's holding register before new data was read into the register.

EOP

Bit 4 notifies the programmer that an operation has been completed.

Interrupt

Bit 2 indicates that an interrupt has been generated by the 1751.

Busy

Bit 1 indicates that the 1751 is in the process of performing an operation.

Ready

Bit 0 indicates that the controller is in a ready state.

12.8.2.2 Director Status II

The programmer may also request the 1751 to send the current sector address of the drum to the lower 12 bits of the A Register. This is accomplished by setting D = 0011_2 and executing an INP instruction.

```
LDQ         =N$0103         EQUIP 2, SECTOR STATUS
NOP
INP         -1              INPUT SECTOR ADDRESS
```

12.8.3 Programming the Drum

In summary, the programmer must first clear interrupts and select desired interrupts. Once this has been issued, the programmer notifies the 1751 of the first word address and the last word address of core memory. Then, he must send the track and sector addresses of the drum. Finally, he specifies the operation to be performed. Status may be taken during the operation to monitor the progress and should be taken at the end of the operation to confirm that the operation was performed correctly.

When data is being written on or read from the drum, the track address is automatically incremented when the sector address overflows to the next track.

Also, the Write Zeros and Check Parity functions operate on a specified area of the drum. Since only a beginning track and sector address were specified, the core address must be sent also to indicate the number of words, even though the data in core is not involved in those operations.

Example:

The following is a test program for the drum. It writes 100 words from a buffer beginning at FWA, on the drum beginning at track 4, sector 0. It then checks drum parity on the data written and reads it back in.

To operate the program, the initial buffer should be set to all one bits from the console. The STOP switch should be set, and the program will stop before the Read. The buffer should then be cleared from the console. By setting the STOP switch again and continuing the RUN, the read will be done and the program will stop. Then the buffer can be swept from the console to see that the data was read.

Note that the drum controller must be dialed to equipment #2 and that the drum address registers and memory address registers must be reset before each operation.

Note also the nifty coding at lines 0026 - 0030 to jump different places on a flag.

12.8.4 Drum Example Program

```
0001                                NAM       DRUM
0002                                ENT       DRUM
0003 P0000 0000     DRUM            0         0
0004 P0001 0A01                     ENA       1          FIRST JMP TO WRITE
0005 P0002 681F                     STA*      FLAG
0006 P0003 0A02                     ENA       $0002      CLEAR CONTROL
0007 P0004 E000                     LDQ       =N$0101    EQUIP 2 DIR FUNC
     P0005 0101
0008 P0006 0B00                     NOP
0009 P0007 03FE                     OUT       -1
0010 P0008 C000     MEM             LDA       =XFWA      BUFFER
     P0009 003E P
0011 P000A E000                     LDQ       =N$010C    FWA(CORE) FUNC
     P000B 010C
0012 P000C 0B00                     NOP
0013 P000D 03FE                     OUT       -1
0014 P000E C000                     LDA       =XFWA+99   LWA
     P000F 00A1 P
0015 P0010 E000                     LDQ       =N$010E    LWA(CORE) FUNC
     P0011 010E
0016 P0012 0B00                     NOP
0017 P0013 03FE                     OUT       -1
0018 P0014 C000     ADDR            LDA       =N$4       TRACK
     P0015 0004
0019 P0016 E000                     LDQ       =N$0108    TRACK ADDR FUNC
     P0017 0108
0020 P0018 0B00                     NOP
0021 P0019 03FE                     OUT       -1
0022 P001A C000                     LDA       =N0        SECTOR
     P001B 0000
0023 P001C E000                     LDQ       =N$010A    SECTOR ADDR FUNC
     P001D 010A
0024 P001E 0B00                     NOP
0025 P001F 03FE                     OUT       -1
0026 P0020 1C00                     NUM        $1C00     JMP*  (0)
0027 P0021 0000     FLAG            0         0          WHERE TO JUMP
0028 P0022 0025 P                   ADC       WRITE
0029 P0023 0033 P                   ADC       CHECK
0030 P0024 0038 P                   ADC       READ
0031 P0025 0A00     WRITE           ENA       0
0032 P0026 E000                     LDQ       =N$0100    WRITE DATA
     P0027 0100
0033 P0028 0B00                     NOP
0034 P0029 03FE                     OUT       -1
0035 P002A E000     STAT            LDQ       =N$0101    DIR FUNC
     P002B 0101
0036 P002C 0B00                     NOP
0037 P002D 02FE                     INP       -1         INP STATUS
0038 P002E 0FCB                     ALS       11         CK EOP
0039 P002F 0131                     SAM        1         SKIP ON EOP
0040 P0030 18FC                     JMP*      STAT+3     WAIT ON EOP
0041 P0031 D8EF                     RAO*      FLAG       NEXT JMP
0042 P0032 18D5                     JMP*      MEM        GO REINITIALIZE
0043 P0033 E000     CHECK           LDQ       =N$0106    CK PAR  ERR ON DRUM (ONLY)
     P0034 0106
```

```
0044 P0035 0B00           NOP
0045 P0036 03FE           OUT        -1
0046 P0037 18F2           JMP*       STAT          GO WAIT ON EOP
0047 P0038 0000    READ   SLS        0                   CLEAR BUFFER FROM CONS
0048 P0039 E000           LDQ        =N$0104       READ FUNC
     P003A 0104
0049 P003B 0B00           NOP
0050 P003C 03FE           OUT        -1
0051 P003D 0000            SLS                     STOP AFTER READ
0052 P003E 0064    FWA    BZS        FWA(100)                   SET BUFFER TO 1'S
0053                      END        DRUM
```

```
 I        00FF  DRUM   0000P MEM    0008P ADDR   0014P FLAG   0021P
 WRITE    0025P STAT   002AP CHECK  0033P READ   0038P FWA    003EP
```

12.9 1731/601 MAGNETIC TAPE

The 1731 magnetic tape controller is used with 601 tape transports. A maximum of eight 601's may be connected to one 1731. Buffering may be accomplished via the 1706.

The 601 is a 7-track transport capable of reading or writing at 200 or 556 Bits Per Inch (BPI). The tape is moved at a rate of 37 1/2 inches per second.

Reading and writing may be done in either Binary or BCD codes. The 601 accepts six bits of data and generates parity for the 7th bit. The parity will be odd for binary and even for BCD.

The data is arranged in groups of records and files. Consecutive frames of information constitute a record. A record may consist of a minimum of one frame. A file is a group of records with the minimum being one record. Longitudinal parity (even) is generated on each recordand stored four spaces past the last data character. A record gap is 3/4' of unrecorded tape surface which denotes the end of a record. A BCD 17_8 code is placed six inches from the last record to indicate the end of a file.

Each time a character is written by a 601, it transfers the character to the 1731 which checks the parity. If the parity is incorrect, the Parity Error status is set and an alarm interrupt is generated. (Note: The alarm interrupt will be generated only if the programmer has selected this interrupt.) The controller also checks for correct parity on a read operation.

The Q Register will be in the following format when programming the 1731.

	15 11	10 7	6 2	1 0
Q	W	E	▨	D

The W field will always be zero when going to the 1731. This field will be used for programming the 1706 which will be discussed later. The E specifies the equipment number of a 1731. The equipment number corresponds with a switch selection on the 1731 ranging from 0 to $F. (Check the equipment setting for your site.) The D field specifies a command.

12.9.1 D = 00 MT Data

This setting specifies a data transfer. A write operation is indicated by an OUT instruction. The write sends the lower six bits of A to the 1731 which generates parity and writes the data and parity on the tape. Whenever the computer breaks the continuity of the character outputs, the controller initiates an End of Record sequence. If no new control functions are issued after the end of record is recorded, tape motion stops.

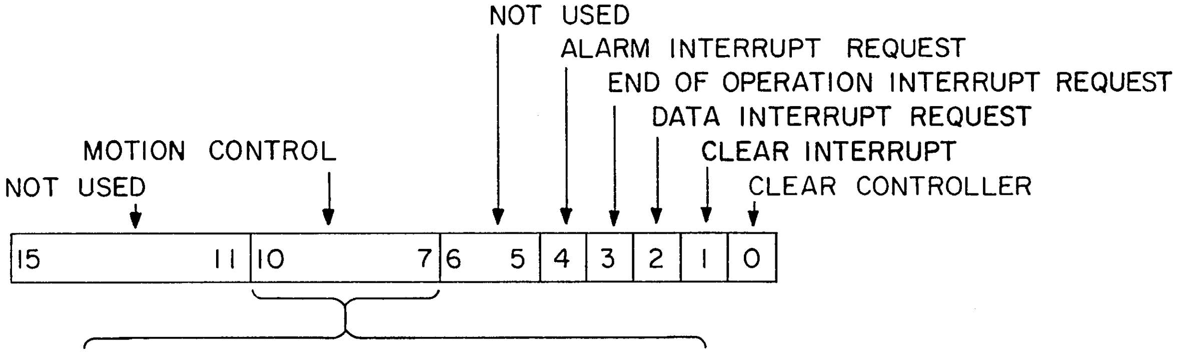
An INP instruction with D = 00 denotes a Read operation. The read operation transfers data from the tape to the controller. The controller checks parity and sends the 6 data bits to the lower 6 bits of the A Register. The 1731 stops sending data to the computer when the computer stops requesting data or when the End of Record is sensed. Tape motion will not terminate except when the End of Record gap is sensed.

If the 1731 is connected to the 1706, the data will be buffered into the computer's memory. The lower six bits of each word in the buffer area will contain data. The A Register will contain the FWA-1 of the buffer area for both read and write operations, therefore, the data transfer will always be initiated with an OUT instruction. The FWA-1 in memory must contain LWA+1 of the buffer area.

Figure 33. 1731 Functions

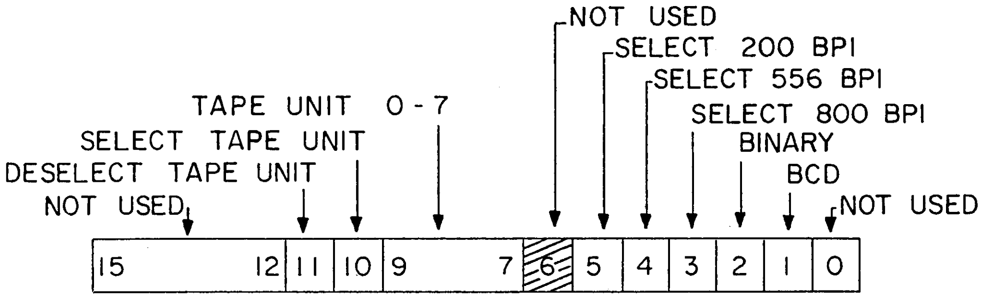
D	COMPUTER INSTRUCTION	
	Output from A	Input to A
00	Write	Read
01	Control Function	Director Status I
10	Unit Select	Director Status II

(A) - Control Function



Bits 10-7 of A	Motion Function
0001	Write Motion
0010	Read Motion
0011	Backspace
0101	Write File Mark
1000	Rewind Load
1100	Rewind Unload

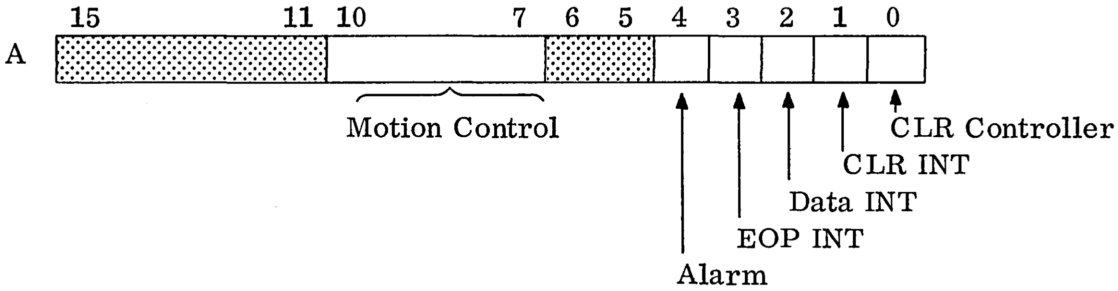
(A) - Unit Select Function



12.9.2 MT Functions

12.9.2.1 D = 01 Control Function

This D setting, accompanied by an OUT instruction, indicates that the A Register contains control functions. The control function gives the programmer the capability of clearing interrupts, clearing the controller, selecting interrupts, and establishing motion control. All control functions may be issued together. The programmer is allowed to select three interrupts: Data, Alarm and End of Operation.



0001 Write Motion

Sets the write logic in the selected 601. Once the logic is set, a data transfer function must be sent with an OUT instruction in order for the data to actually be written on tape.

0010 Read Motion

Sets the read logic within the selected 601. A data transfer function must be sent with an INP instruction in order for the data to be transferred to the A Register.

0011 Backspace

Causes the 601 to backspace one record.

0101 Write File Mark

Write file mark generates six inches of blank tape followed by a 17_8. When the end of file mark is written or read, longitudinal parity is checked. If the controller is in binary mode, a parity error will be generated, as the 17_8 is in BCD mode (even parity).

1000 Rewind Load

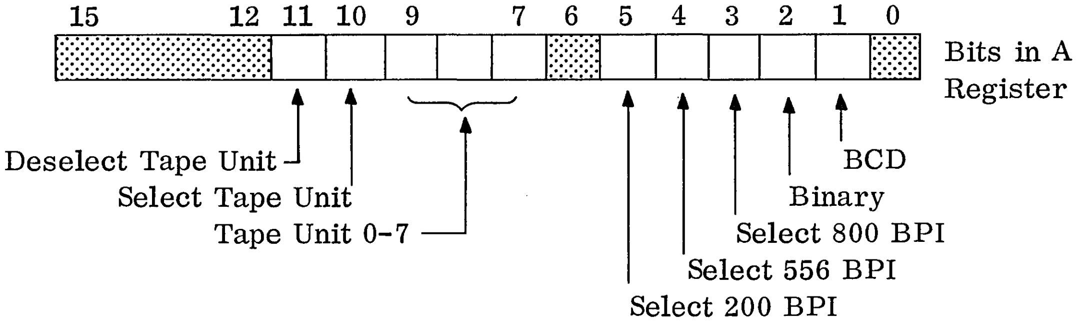
The rewind controls bring the tape back to the magnetic load point indicator. The ready status stays up without any manual intervention.

1100 Rewind Unload

The rewind load keeps the ready status while the rewind unload causes the 601 to drop ready.

12.9.2.2 D = 10 Unit Select Function

Allows the programmer to select the desired 601, density, and mode, when accompanied by an OUT instruction. The 601 tape units can read at 200 and 556 BPI. The deselect function, bit 11, is used to deselect a protected 601 in order that an unprotected program may·get access to the 1731 to use an unprotected 601.

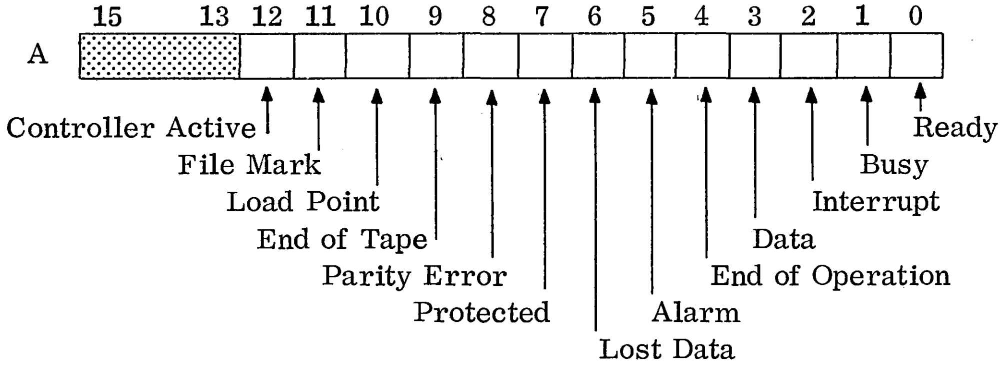


The Select Tape Unit, Bit 10, indicates that the unit number in Bits 9-7 is to be the desired unit. If Bits 10 and 11 are not set, the controller ignores Bits 7-9.

12.9.3 MT Status

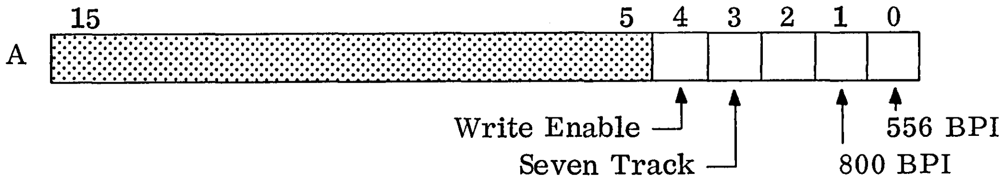
12.9.3.1 D = 01 Status I

This D setting brings Director Status I into the A Register when accompanied with an INP instruction.



12.9.3.2 D = 10 Status II

Director Status II is requested with this D setting on an INP instruction.



12.9.4 Magnetic Tape Example Programs

12.9.4.1 MT Example 1

The following is a test program for a 601 magnetic tape on a 1731 controller. (It could also be used for a 608 tape on a 1732 controller.) The program generates 200 frames of data in the A Register and outputs 1 frame at a time (the lower order 6 bits of A) to the tape unit. The program then backspaces the tape and reads the data back in, storing it in the buffer DATA. Each frame of data occupies the lower 6 bits of a word in the buffer.

The program canbe run with the STOP switch set. It will stop when finished, and the buffer canbe swept from the console. One cannot "step through" the program because the tape is moving as soon as the first tape motion command is issued.

```
           NAM      MT601
           LDQ      =N$0382         Q382      EQUIP 7, UNIT SEL
USEL       LDA      =N$0494                   UNIT1,556BPI,BIN
           NOP
           OUT      -1
           INQ      -1              Q381      MOTION FUNC
WRITEMO    LDA      =N$81                     WRITE MOT, CLR CONT.
           NOP
           OUT      -1
           LDA      =N-200                    200 FRAMES
           STA-     I
           INQ      -1              Q380      DATA FUNC
DATA       LDA      =N$FF                     DATA 3F
           NOP
           OUT      -1
           RAO-     I
           LDA-     I
           SAZ      BACKSP
           JMP*     DATA+3
BACKSP     INQ      1               Q381      MOTION FUNC
           LDA      =N$0180                   BACKSPACE
           NOP
           OUT      -1
READMO     LDA      =N$0100                   READ MOTION
           NOP
           OUT      -1
           LDA      =N-200
           STA-     I
           INQ      -1
RDDATA     NOP                      Q830      DATA FUNC
           INP      -1
           STA*     DATA+200,I                STORE DATA
           RAO-     I
           LDA-     I
           SAZ      STOP-*-1
           JMP*     RDDATA+1
STOP       SLS      0                         SWEEP BUFFER TO CHECK
DATA       BZS      DATA(200)
           END
```

12.9.4.2 MT Example 2 - With Error Checks

This test program for magnetic tape is the same as Example 1, with the addition of error checks. Note that the program never hangs in a loop on a reject.

```
NOP
OUT             -1
```

Instead it jumps to REJINT for any internal reject or REJEXT for any external reject. Even with no error analysis, note that considerably more coding is required just to allow for errors. Also note that the program is continually waiting for the tape.

The program will stop either on normal termination or after a reject. It can be restarted after a reject by simply correcting the error condition and setting the RUN switch.

This particular program has been run with the 1706 code set but it is still an unbuffered operation. The site where it was run had their magnetic tapes connected through the 1706, but the Direct Storage Access line was not also connected so operations simply went through the 1706 in unbuffered mode.

```
          0001                               NAM    TEST        1706
          0002 P0000 E000    A               LDQ    =N$1382     EQUIP 7
               P0001 1382                                       UNIT SEL
          0003 P0002 C000                    LDA    =N$0494
Select         P0003 0494                                       UNIT 1, %%¢ BPI, BINARY
          0004 P0004 0302                    OUT    2
          0005 P0005 1804                    JMP*   A2          GOOD
          0006 P0006 586A                    RTJ*   REJINT      INTERNAL REJECT
          0007 P0007 5861                    RTJ*   REJEXT      EXTERNAL REJECT
          0008 P0008 18F7                    JMP*   A           RETURN AFTER REJECT
          0009 P0009 E000    A2              LDQ    =N$1381
               P000A 1381                                       CONTROL FUNCTION
          0010 P000B C000                    LDA    =N$0081
/rite          P000C 0081                                       WRITE MOT - CLR CONTROL
/lotion   0011 P000D 0302                    OUT    2
          0012 P000E 1804                    JMP*   A4
          0013 P000F 5861                    RTJ*   REJINT
          0014 P0010 5858                    RTJ*   REJEXT
          0015 P0011 18F7                    JMP*   A2
          0016 P0012 C000    A4              LDA    =N-200      EOR 200 FRAMES
               P0013 FF37
          0017 P0014 60FF                    STA-   I
          0018 P0015 E000    A5              LDQ    =N$1381     DIR STAT 1
               P0016 1381
          0019 P0017 0202                    INP    2           STATUS
Status    0020 P0018 1804                    JMP*   A55
          0021 P0019 5857                    RTJ*   REJINT
          0022 P001A 584E                    RTJ*   REJEXT
          0023 P001B 18F9                    JMP*   A5
          0024 P001C 0FCC    A55             ALS    12          DATA READY?
          0025 P001D 0131                    SAM    *+2
Data      0026 P001E 18F6                    JMP*   A5
Ready?    0027 P001F 0DFE                    INQ    -1          WRITE DATA FUNC
          0028 P0020 C000                    LDA    =N$FF
               P0021 00FF                                       0000 0000 1111 1111
          0029 P0022 0305    OUT             OUT    5
          0030 P0023 D0FF                    RAO-   I
          0031 P0024 C0FF                    LDA-   I
Output    0032 P0025 0104                    SAZ    A7-*-1
          0033 P0026 18EE                    JMP*   A5          GO STATUS AGAIN FOR DATA READY
Data      0034 P0027 5849                    RTJ*   REJINT
          0035 P0028 5840                    RTJ*   REJEXT
          0036 P0029 18EB                    JMP*   A5
          0037 P002A E000    A7              LDQ    =N$1381     DIR STATUS 1
               P002B 1381
          0038 P002C 0205                    INP    5
          0039 P002D A000                    AND    =N$2        CHECK BUSY
               P002E 0002
Busy?     0040 P002F 0104                    SAZ    A9-*-1      SKIP WHEN NOT BUSY
          0041 P0030 18F9                    JMP*   A7
          0042 P0031 583F                    RTJ*   REJINT
          0043 P0032 5836                    RTJ*   REJEXT
          0044 P0033 18F6                    JMP*   A7
          0045 P0034 E000    A9              LDQ    =N$1381     CONTROL FUNCTION
               P0035 1381
```

12.9.4.2

```
            0046 P0036 C000                 LDA      =N$0180         BACKSPACE
                 P0037 0180
            0047 P0038 0302                 OUT      2
Backspace   0048 P0039 1804                 JMP*     A11
            0049 P003A 5836                 RTJ*     REJINT
            0050 P003B 582D                 RTJ*     REJEXT
            0051 P003C 18F7                 JMP*     A9
            0052 P003D E000  A11            LDQ      =N$1381         DIR STATUS 1
                 P003E 1381
            0053 P003F 0202              INP    2
            0054 P0040 1804                 JMP*     A111
            0055 P0041 582F                 RTJ*     REJINT
            0056 P0042 5826                 RTJ*     REJEXT
Busy?       0057 P0043 1801                 JMP*     A111
            0058 P0044 0FCE  A111           ALS      14              BUSY?
            0059 P0045 0121                 SAP      *+2             SKIP WHEN NOT BUSY
            0060 P0046 18F6                 JMP*     A11
            0061 P0047 C000      LDA      =N$0100                    SEL READ MOTION
                 P0048 0100
Read        0062 P0049 0302                 OUT      2
Motion      0063 P004A 1804                 JMP*     A13
            0064 P004B 5825                 RTJ*     REJINT
            0065 P004C 581C                 RTJ*     REJEXT
            0066 P004D 18EF                 JMP*     A11
            0067 P004E C000  A13            LDA      =N-200
E-7              P004F FF37
            0068 P0050 60FF                 STA-     I
            0069 P0051 E000  A14             LDQ      =N$1381        STATUS
                 P0052 1381
            0070 P0053 0202                 INP        2
            0071 P0054 1804                 JMP*     A44
            0072 P0055 581B                 RTJ*     REJINT
Data        0073 P0056 5812                 RTJ*     REJEXT
Ready?      0074 P0057 18F9                 JMP*     A14
            0075 P0058 0FCC  A44            ALS      12              WAIT FOR DATA READY
            0076 P0059 0131                 SAM      *+2
            0077 P005A 18F6                JMP*      A14
            0078 P005B 0DFE                 INQ      -1              INPUT DATA
            0079 P005C 0A00                 ENA      0
            0080 P005D 0207                 INP      7
            0081 P005E 6900                 STA      DATA+200,I      STORE DATA NOT
Input            P005F 00E2                                            ASSEMBLED
Data        0082 P0060 D0FF                 RAO-     I
            0083 P0061 C0FF                 LDA-     I
            0084 P0062 0104                 SAZ      A16-*-1
            0085 P0063 18ED                 JMP*     A14
            0086 P0064 580C                 RTJ*     REJINT
            0087 P0065 5803                 RTJ*     REJEXT
            0088 P0066 18EA                 JMP*     A14
            0089 P0067 0000  A16            SLS                      STOP WHEN THROUGH
            0090 P0068 0000  REJEXT         0        0
External    0091 P0069 E000                 LDQ      =N$1381         EXT REJ
Reject           P006A 1381
            0092 P006B 0B00                 NOP                      TAKE STATUS
            0093 P006C 02FE                 INP      -1              AND STOP
```

```
0094 P006D E8FA                LDQ*      REJEXT
0095 P006E 0000                SLS      0               RUN TO RETURN
0096 P006F 1CF8                JMP*     (REJEXT)
0097 P0070 0000   REJINT       0        0
0098 P0071 E000                LDQ      =N$1381         INTERNAL REJECT
     P0072 1381
0099 P0073 0B00                NOP                      TAKE STATUS
0100 P0074 02FE                INP      -1              AND STOP
0101 P0075 E8FA                LDQ*      REJINT         RUN TO RETURN
0102 P0076 D8F9                RAO*     REJINT
0103 P0077 0000                SLS      0
0104 P0078 1CF7                JMP*     (REJINT)
0105 P0079 00C8                BZS      DATA(200)       DATA BLOCK
0106                           END
```

12.10 1732/608-609 MAGNETIC TAPE

The 1732 controller for 608 and 609 Magnetic tapes is very similar to the 1731 and can be programmed in identically the same way as the 1731. For that reason, a separate program for the 1732 is not included here. Instead, the 1731 program was run on the 1732/608.

The 1732 provides an additional feature which was not available on the 1731: option for selecting assembly/disassembly mode. This means that two frames at a time can be sent to or received from the controller in one OUT or INP (consequently meaning the controller has to be accessed half as often). The controller takes care of assembling or disassembling the frames on the tape. Bit 6 (not used on the 1731) in the function code is used to select this mode.

This is especially useful on the 609 (9-track tape) in that two 8-bit frames exactly fit in one 16-bit 1700 word. Repacking the buffer can be eliminated since a word at a time is sent to the controller. The 609 uses only 800 BPI density, and normal end of files are not used on it. For example:

- Load 601 MT program from lockheed (or reassemble with changes)
- Clear 1706 code from all Q addresses - MT's are not on 1706 (i.e., change $1382 to $0382)
- Change P0003 to:
  $4D4 for 608 (adds selection of assembly)
  $4CC for 609 (800 BPI only, assembly)
- Change P0021 to FFFF
- Follow operating instructions on 601 program
- Output from A will be (in assembly/disassembly):

608 - bits 8-13 and 0-5	XX111111	XX111111
609 - bits 8-15 and 0-7	11111111	11111111

- Input to A will be:

608	00111111	00111111
609	11111111	11111111

- The upper bits in A are the first frame; the lower bits are the second frame.

12.11 1706 BUFFER DATA CHANNEL

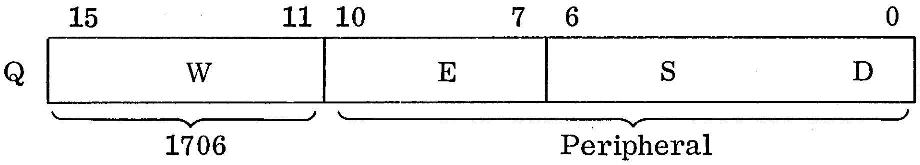
The 1706 is a 16-bit, bidirectional, buffer channel with word transfer rates up to 900 KC (approximately 1.1 microseconds per 16-bit word).

The 1706 buffers data between the computer's memory and a peripheral. The 1706 is capable of buffering as many as eight devices. The 1700 system may have three 1706's attached.

The 1706 has no indicators nor control panels, therefore, all operations are initiated by the computer via the A/Q channel. The 1706 is considered as one of the eight devices attached to the A/Q channel and the DSA channel with each peripheral connected to the 1706 being a substation. Consequently, only one of the 1706's peripherals may be referenced at a time.

The program requests direct access to a peripheral via the 1706 to establish the logic. Once the logic has been established, the program requests the 1706 to perform the data transfer.

Bits 11-15, the W field, of the Q Register are used to reference the 1706 and to indicate the desired operation. Bits 0-10 of the Q Register will contain the same bit setting used to reference a particular peripheral when it is not attached to the 1706.



It is possible to perform direct I/O on a device connected to the 1706 simply by setting W to 00010 and sending all the codes for the device thru the 1706. (However, normally if a device is on the 1706, it is desired to perform data transfers in a buffered mode by letting the 1706 perform the operations.)

12.11.1 1706 Functions

The setting of the W field is dependent upon which of the possible three 1706's the program is referencing. There are four settings for each 1706.

	W SETTING*			COMPUTER OPERATION	
	1706 #3	1706 #2	1706 #1	INP	OUT
1.	$_0$C	$_0$7	$_0$2	Direct Input	Direct Output
2.	$_0$D	$_0$8	$_0$3	Terminate Buffer: Current Addr of 1706	Function
3.	$_0$E	$_0$9	$_0$4	1706 Status	Buffered Output
4.	$_0$F	$_0$A	$_0$5	1706 Current Addr	Buffered Input

*The left digit is binary, the right digit is hexadecimal.

The first W setting provides the computer direct access to the peripheral. The peripheral may be requested to send to the A Register a data word or status word. The computer may send to the peripheral a function or data word from the A Register. This mode of operation is identical in every way to that on the A/Q channel.

12.11.2 Programming the Peripheral Through the 1706

The 609 magnetic tape unit shall be used as an example with 1706 number 1. It is necessary to set up the equipment prior to telling the 1706 to do any I/O on the equipment.

```
LDQ        =N$1202        DIRECT OUT

LDA        =N$4CC         SEL UNIT 1, ASSEMBLY,
                          800 BPI, BIN

NOP

OUT        -1

INQ        -1             PREPARE FOR TAPE MOTION

LDA        =N$80          WRITE MOTION

NOP

OUT        -1
```

The 609 tape unit has now been functioned. The next step is to function the 1706, requesting it to interrupt the computer when the data transfer is complete. (This is if the 1706 will be operated in interrupt mode.)

```
LDQ        =N$1800        1706 FOR FUNCTION #3

LDA        =N$8001        INT ON EOP

NOP

OUT        -1             1706 IS FUNCTIONED
```

The 1706 and the peripheral have both been functioned. The next step is to initiate the I/O operation. At this point the 1706 will take over and do the data transfer. It will now be impossible to directly access the peripheral until the 1706 is finished or becomes hung up.

The 1706 expects to find the First Word Address minus one (FWA-1) of the buffer area in the A Register when the I/O operation is initiated. Upon receiving the FWA-1 the 1706 goes into the computer's memory and extracts the Last Word Address plus one (LWA+1) from that location. The 1706 then updates the FWA-1 by 1 until it equals the LWA+1 at which point the data transfer is complete.

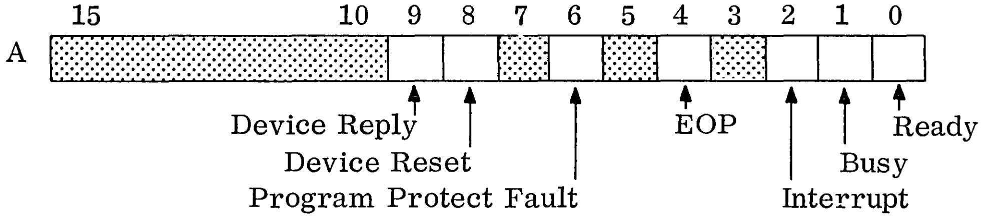
```
        LDA     =XLWA+1     LAST WORD ADDR + 1 IN A
        STA     FWA-1       LWA=1 AT FWA-1
        LDQ     =N$2200     BUFFER OUT.  EQUIP #4
        LDA     =XFWA-1     FWA-1 IN A
        NOP
        OUT     -1          OPERATION INITIATED
```

The program at this point may exit and wait for the End of Operation interrupt. Two other alternatives are available: status for End of Operation or status for current address.

```
        LDQ     =N$2200     STATUS 1706

        NOP

STAT    INP     -1          STATUS IN A

        ALS     11          EOP BIT AT SIGN BIT

        SAM     CMP         WHEN SET OP COMPLETE

        JMP*    STAT        WAIT UNTIL COMPLETE

CMP     -       -

        LDQ     =N$2A00     CURRENT ADDR,  EQUIP 4

        NOP

STADR   INP     -1          CURRENT ADDR IN A

        SUB     =XLWA+1     SUBT LWA+1

        SAZ     CMP1        ZERO,  OPERATION CMP

        JMP*    STADR       CONTINUE STATUS FOR ADDR

CMP1
```

12.11.3 1706 Status

Once the data transfer is complete, the program may process the data. The program may at any time status the 1706 for the current address and for the 1706 status word. The program may check the status word for the following information.



Note that this is the status of the 1706, not the peripheral. It is not possible to get the status of the peripheral while the 1706 is working on it.

Ready (Bit 0 = 1) → This bit is set when power is on.

Busy (Bit 1 = 1) → This bit is set from the time the 1706 accepts an output word from the computer initiating a block transfer until the block transfer is terminated, or during a direct operation.

Interrupt (Bit 2 = 1) → A buffer transfer input or output has been completed.

Program protect fault (Bit 6=1) → A reference to computer storage caused a program protect fault.

Device Reject (Bit 8 = 1) → This bit, if set, means the peripheral device rejected the last word transfer attempted from the 1706.

Device Reply (Bit 9 = 1) → This bit, if set, means the peripheral device accepted the last word transfer attempted from the 1706.

It is possible for the 1706 to get hung up as it continually repeats an attempt to make a data transfer to the peripheral if the peripheral fails.

The program may status and find a Device Reject status. If this condition were to arise, the program may terminate the buffer operation. This termination is always necessary when the buffer becomes hung up. When the operation is terminated, the current address is sent to the A Register automatically.

```
LDQ      =N$1A00     TERMINATE BUFFER, EQUIP 4
NOP
INP      -1          CURRENT ADDR IN A
```

12.11.4 Summary of 1706

In summary, the computer functions the peripheral direct via the 1706. Once the peripheral is functioned, the End of Operation interrupt is requested. The program must have the LWA+1 at the FWA-1 prior to initiating a buffer operation. When the buffer operation is initiated, the FWA-1 is in the A Register and sent to the 1706. The status word of the 1706 may be requested anytime as well as the current address. The program cannot status the peripheral itself until the operation is completed or terminated.

12.11.5 1706 Example Program

Example:

The following is a test program for a 609 magnetic tape on a 1732 controller, operated in buffered mode by the 1706. The program outputs 50 words of data from the buffer beginning at BUF+1, rewinds the tape, and reads the data back in.

The buffer should be set to all one bits from the console. Then the program should be operated with the STOP switch set so that the program will stop after writing (to allow the programmer to clear the buffer from the console). After reading, it will stop again where the buffer can be swept to see the data.

12.11.5

```
0001                          NAM       MT609
0002                          ENT       MT
0003       1000               EQU       OUT06($1000)      DIRECT OUT
0004       1800               EQU       FUN06($1800)      FUNCTION
0005       2000               EQU       BUF06($2000)      BUFFER OUT OR STATUS
0006       2800               EQU       BUFIN($2800)      BUFFER IN
0007       0280               EQU       EQUIP($280)       1732 EQUIP #4
0008                *
0009                *SET BUFFER TO ONE,S FROM CONSOLE
0010                *SET STOP SWITCH
0011                *
0012                *
0013 P0000 E000     MT        LDQ       =XOUT06+EQUIP+1   DIRECT OUT
     P0001 1281
0014 P0002 0A01               ENA       1                 CLR CONTROLLER
0015 P0003 03FE               OUT       -1
0016 P0004 E000               LDQ       =XOUT06+EQUIP+2   DIRECT OUT
     P0005 1282
0017 P0006 C000               LDA       =N$4CC            SEL UNIT 1, ASSEMBLY,
     P0007 04CC                                               800 bpi, BINARY
0018 P0008 0B00               NOP
0019 P0009 03FE               OUT       -1
0020 P000A 0DFE               INQ       -1
0021 P000B C000               LDA       =N$80             WRITE MOTION
     P000C 0080
0022 P000D 0B00               NOP
0023 P000E 03FE               OUT       -1
0024                *                                   HERE IS WHERE NORMALLY WE
0025                *                                   WOULD SEL EOP INT AND
0026                *                                   OUTPUT IT
0027 P000F E000               LDQ       =XBUF06+EQUIP     BUFFER OUT
     P0010 2280
0028 P0011 C000               LDA       =XBUF             FWA-1
     P0012 002C P
0029 P0013 0B00               NOP
0030 P0014 03FE               OUT       -1
0031 P0015 0B00     STAT      NOP
0032 P0016 02FE               INP       -1                INPUT STATUS
0033 P0017 0FCB               ALS       11                WAIT FOR EOP
0034 P0018 0131               SAM       *+2               EOP IMPLIES GOOD PARITY
0035 P0019 18FB               JMP*      STAT
0036 P001A 0000             SLS         0                 CLEAR BUF FROM CONSOLE
0037 P001B E000               LDQ       =XOUT06+EQUIP+1   DIRECT OUT
     P001C 1281
0038 P001D C000               LDA       =N$400            REWIND TAPE
     P001E 0400
0039 P001F 0B00               NOP
0040 P0020 03FE               OUT       -1
0041 P0021 C000               LDA       =N$100            READ MOTION
     P0022 0100
0042 P0023 0B00               NOP
0043 P0024 03FE               OUT       -1
0044 P0025 E000               LDQ       =XBUFIN+EQUIP     BUFFER IN
     P0026 2A80
0045 P0027 C000               LDA       =XBUF             FWA-1
     P0028 002C P
0046 P0029 0B00               NOP
0047 P002A 03FE               OUT       -1
0048 P002B 0000               SLS       0
0049 P002C 0033     BUF       RZS       BUF(51)           1 CONTROL WORD,
0050       002C P             ORG       BUF                   50 DATA WORDS
0051 P002C 005F P             ADC       BUF+51            PUT LWA+1 IN FWA-1
0052       005F P             ORG*
0053                          END
```

CHAPTER XIII

PERIPHERAL PROGRAMMING - II

13

CHAPTER XIII - Peripheral Programming II

13.0 INTRODUCTION

The preceding chapter, Peripheral Programming I, discussed in detail the procedures for programming the standard 1700 peripherals. Inefficiency would result if each user program were required to contain the coding necessary to drive a peripheral; therefore, the operating system contains programs that perform all input/output operations. These programs are referred to as drivers.

Drivers are divided into three main parts: initiator, continuator and error.

13.1 INITIATOR SECTION OF DRIVER

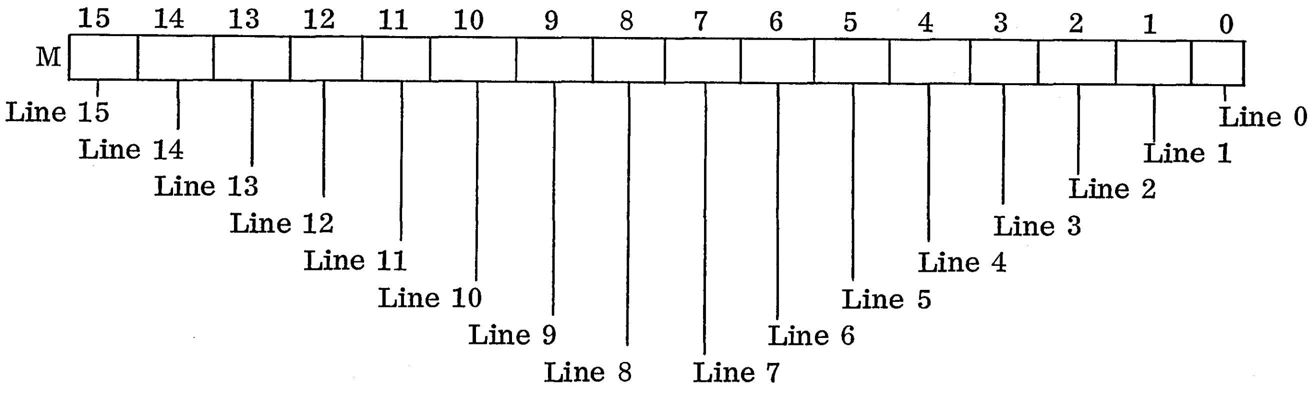
The initiator portion sets up the logic to be used and initiates the operations to be performed. The paper tape reader shall be used as an example. All of the drivers are written in interrupt mode. Interrupt mode allows the driver to initiate an operation and select interrupts, exit to the operating system and regain control when the peripheral has completed the operation.

```
LDQ        =N$A1       SEL PTR, FUNC
ENA        1           CLR CONTROLLER
OUT        -1
ENA        $34         START MOTION, SEL INT ON
OUT        -1          ALARM OR DATA
EXIT TO OPERATING SYSTEM
```

Via the above coding, the initiator portion of the paper tape reader has selected the equipment, selected interrupts, and exited to wait for an interrupt. The initiator portion of every driver initiates these three operations.

13.2 INTERRUPT FROM EQUIPMENT

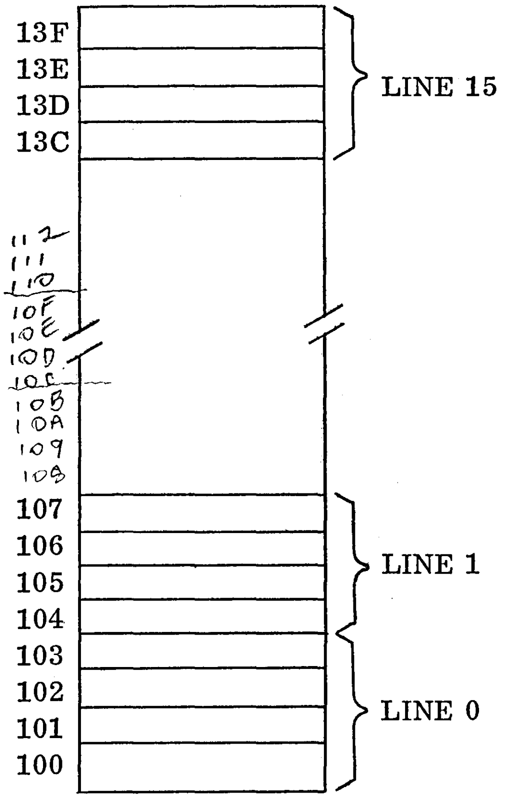
The equipment will generate an interrupt when a selected interrupt condition arises. The acknowledgement of interrupt is on a priority basis. The priority depends on the setting of the 16-bit mask register. A maximum of 16 interrupt lines may be connected to the 1700, with each line corresponding to a bit in the mask register.



13.2

The bit in the corresponding bit position must be a 1 in order for the interrupt to be acknowledged. If the bit is a 0, the interrupt holds and is not acknowledged until the bit becomes a 1.

Once the bit is set to a 1 in the M register, control is transferred to the interrupt trap region. The trap region is set up to allow four words for each interrupt line. The core locations are always from location 100_{16} to $13F_{16}$. See section 1.3 to review the interrupt system.



Interrupts may be nested 16 deep!

Four core locations reserved for each interrupt line:

- word 4 - address of interrupt processor
- word 3 - priority level for line
- word 2 - RTJ to interrupt handler
- word 1 - overflow and P

Figure 34. 1700 Interrupt Hardware and Software Functions

Hardware:

- Disables interrupts
- Stores overflow indicator and P of interrupted program in word 1
- Transfers control to word 2

Software:

- Word 2 contains RTJ to common interrupt handler
- Interrupt handler saves registers of interrupted program, sets new mask from priority level in word 3, enables interrupts, and transfers control to interrupt processor for that line (from address in word 4).
- Interrupt processor must exit to the driver continuator which will service the equipment (i.e., input data).
- The continuator must exit through the dispatcher to restore the interrupted program.

The computer hardware disables interrupts and stores the contents of the P register in the lower 15 bits of the first trap word for the interrupting line. The hardware sets the upper bit of the first word to a 1 if the overflow indicator is on and to a 0 if the indicator is off. Control is then passed to the second word.

Once control is passed to the second word, the processing is under software control. See Figure 36 for hardware and software functions.

13.3 COMMON INTERRUPT HANDLER

The second word contains a return jump to the common interrupt handler. The common interrupt handler saves the contents of all the pertinent registers: A, Q, M, and I. The M register is set to the priority for the interrupting line by using the priority level set in the third word. Interrupts are then enabled by the common interrupt handler. The interrupt handler transfers control indirectly through the fourth word to the processor for that line.

13.4 INTERRUPT LINE PROCESSOR

The processor for the interrupt line (LYNEI or EPROC) takes status on all equipment on the interrupting line and checks bit 2 of each status word.

The processor will be able to determine which peripheral interrupted because the interrupt bit (bit 2) of the status word will be set. The interrupt processor will pass control to the appropriate continuator portion of a driver.

13.5 CONTINUATOR SECTION OF DRIVER

The continuator checks the alarm bit to determine if control should be passed to the error portion of the driver.

13.5

```
          LDQ       =N$A1       SEL PTR, STATUS
          NOP
          INP       -1          STATUS IN A
          STA       STATUS      SAVE STATUS
          AND       =N$20       CK FOR ALARM
          SAZ       1           IF ZERO CONTINUE
          JMP       ERR         IF NOT ZERO ALARM
```

If the alarm bit were not set the interrupting condition would be processed by the continuator. The continuator checks further to determine which interrupt was generated. The paper tape reader allows the selection of only two interrupts, alarm and data; therefore, if the alarm bit was not set the data interrupt was probably generated. It would be wise to check the data interrupt bit and if it is not set, pass control to GI (ghost interrupt) in the error section:

```
          LDA       STATUS
          AND       =N8
          SAN       DATA
          JMP       GI
```

Input data if the data bit was set:

```
DATA      LDQ       =N$AO       SEL PTR, DATA
          NOP
          INP       -1          DATA IN A
```

The continuator then performs the necessary packing operations to form one 16-bit word. A check is made to determine if all data has been processed. If not, the continuator exits to wait for the next interrupt.

```
          LDQ       =N$A1       SEL PTR, FUNC
          ENA       $14         INT ON DATA OR ALARM
          OUT       -1
          EXIT TO OPERATING SYSTEM
```

The continuator could simply exit without reselecting interrupts because the interrupt request is still up if it has not been cleared.

13.6 ERROR SECTION OF DRIVER

The ERROR portion takes status to determine which alarm condition has arisen. The error routine then performs the necessary operation to correct the error. If the error cannot be corrected without operator intervention, the operator should be notified.

```
LDQ      =N$A1     SEL PTR, STATUS
NOP
INP      -1        STATUS IN A
ALS      5         POWER ON BIT AT 15
SAM      POWOFF
ALS      1         PAPER MOTION FAIL BIT AT 15
SAP      PMF
ALS      3         LOST DATA BIT AT 15
SAM      LOSTD
JMP      GI        IF HERE NO ALARM OCCURRED
```

The above coding establishes the condition at fault. The skip address sends control to routines that process the various errors.

GI is where control is passed for a ghost interrupt. The equipment interrupted but apparently for no reason. This would indicate a hardware malfunction.

13.7 SUMMARY

The driver can be summarized as follows. The initiator is the first to have control. It selects the equipment and selects interrupts, then it exits to the operating system. The continuator gains control via the interrupt trap area after the peripheral generates an interrupt. It checks for alarm and if one is present, sends control to the error portion. If no error occurred, it maintains control and processes the interrupt. If the operation is not complete the continuator exits to the operating system. The error portion determines which alarm condition occurred and attempts to correct the fault and/or notifies the operator. Figure 35 illustrates the flow of the interrupts through the continuator.

It is important to note that the primary purpose of this chapter is to illustrate techniques for programming the hardware in interrupt mode. The linkage through MSOS routines is secondary.

The logical division of programming functions into initiator, continuator, and error sections could be utilized to program any peripheral, either in a stand-alone system or under MSOS. The operations to be included in each section are the important consideration here -- what the equipment is capable of doing, its timing, and the status of responses it can send. At this point the programmer should be able to write an interrupt-mode driver which would not run under MSOS.

To actually write a driver to run under MSOS, it would be necessary to study all the linkage to MSOS since system tables and common subroutines are used by all the MSOS drivers.

Some of the MSOS routines are included at the end of this chapter for illustration.

13.7

Example

The following is a test program to print a message on the teletype in interrupt mode. It uses the MSOS Interrupt Handler to save the state of the interrupted program (probably the idle loop) each time the interrupt comes in. It bypasses the Line 1 Interrupt Processor (interrupt response routine) and MSOS driver by storing its own address in the fourth word of the interrupt trap for Line 1 (location $107).

After the program has been assembled and loaded under MSOS, the computer should be stopped, the protect switch turned off, and P set to the address TTYI; then it should be run. A master clear should not be done because that would disable the interrupt system and clear M.

In reality, if a program such as this were used in a stand alone system, it would assure that bit 1 was set in the M register and execute an EIN. It would also have a routine corresponding to the interrupt handler to save the state of an interrupted program.

The test routine here simulates the operations on the equipment which would logically be performed by different portions of a driver.

1. TTYI sets up the trap.

2. INIT is the initiator to set up the equipment. It then exits and waits for the first interrupt.

3. CONT is the continuator and it outputs a character each time the interrupt comes in. It also must keep track of the number of words desired to be written. It hangs at CMPLET when finished. An MSOS driver would schedule the programmer's completion address when his request was finished.

4. ERR. The error section analyzes errors. It hangs in the test routine on each error, but in MSOS it would attempt to correct the error.

5. The TABLES used by the routine contain information which is used by the driver for the write; they simulate a physical device table.

6. INTRES. The interrupt response routine is actually not a part of the driver. It must status each device on the line to see which one interrupted. Our example only checks the TTY.

```
0001                        NAM       TTY INTERRUPT MODE
0002                  * LOAD THE PROGRAM UNDER MSOS - TURN OFF PROTECT SWITCH
0003                  * TURN OFF DISC - DO NOT MASTER CLEAR - SET P AND RUN
0004       00EA             EQU       ADISP($EA)
0005                        ENT       TTYI,CONT,ERR,INTRES
0006 P0000 0000  TTYI       0         0
0007 P0001 C000             LDA       =XINTRES      SUBSTITUET INTRES IN TRAP
     P0002 003A P
0008 P0003 6400             STA+      $107
     P0004 0107
0009                  *
0010                  *                         INITIATOR
0011 P0005 E830  INIT       LDQ*      TTY           GET TTY FUNC CODE
0012 P0006 0A03             ENA       3             CLR CONTROL  CLR INTERRUPT
0013 P0007 03FE             OUT       -1
0014 P0008 C000             LDA       =N$100        SELECT WRITE MODE
     P0009 0100
0015 P000A 0B00             NOP
0016 P000B 03FE             OUT       -1
0017 P000C 0DFE             INQ       -1            CHANGE TO DATA FUNCTION
0018 P000D 0B00             NOP
0019 P000E 03FE             OUT       -1            OUTPUT DUMMY CHARACTER
0020 P000F 0D01             INQ       1             SEND FUNCTION
0021 P0010 0A14             ENA       $14           SEL INTERRUPT ALARM OR DAT
0022 P0011 03FE             OUT       -1
0023 P0012 14EA             JMP-      (ADISP)       GO WAIT FOR INT
0024                  *
0025                  *                         CONTINUATOR
0026 P0013 C825  CONT       LDA*      STAT          GET STATUS BACK
0027 P0014 0FCA             ALS       10            CK ALARM BIT 5
0028 P0015 0121             SAP       OK-*-1
0029 P0016 1817             JMP*      ERR           ALARM UP
0030 P0017 0FC2  OK         ALS       2             CK DATA BIT 3
0031 P0018 0131             SAM       DATA
0032 P0019 1827             JMP*      GI            NOT ALARM OR DATA
0033 P001A CC00  DATA       LDA       (FWA)         GET DATA
     P001B 001B
0034 P001C E81D             LDQ*      FLAG          GET CHAR FLAG
0035 P001D 0144             SQZ       LOWER
0036 P001E 0FC8             ALS       8             UPPER CHAR
0037 P001F 0C00             ENQ       0
0038 P0020 4819             STQ*      FLAG          CLR FLAG FOR LOWER NEXT
0039 P0021 1803             JMP*      OUTPUT        GO OUTPUT DATA
0040 P0022 D817  LOWER      RAO*      FLAG          SET FLAG FOR UPPER NEXT
0041 P0023 D813             RAO*      FWA           UPDATE BUFFER ADDR
0042 P0024 E811  OUTPUT     LDQ*      TTY
0043 P0025 0DFE             INQ       -1            DATA FUNC
0044 P0026 0B00             NOP
0045 P0027 03FE             OUT       -1            OUTPUT 1 CHAR
0046 P0028 C80E             LDA*      FWA           LAST WORD YET
0047 P0029 B80E             EOR*      LWA
0048 P002A 0101             SAZ       CMPLET
0049 P002B 14EA             JMP-      (ADISP)       GO AWAIT NEXT INT
0050 P002C 18FF  CMPLET      NUM      $18FF         HANG WHEN FINISHED
0051                  *
```

```
0052                 *                                ERROR SECTION
0053                 *                                FIND CAUSE OF ERROR
0054 P002D C808      ERR      LDA*     STAT
0055 P002E 0FC5               ALS      5                 CK MOTOR ONN
0056 P002F 0131               SAM      1
0057 P0030 18FF               NUM      $18FF             MOTOR OFF - HANG HERE
0058 P0031 0FC4               ALS      4                 CK LOST DATA
0059 P0032 0121               SAP      1
0060 P0033 18FF               NUM      $18FF             LOST DATA - HANG HERE
0061 P0034 180C               JMP*     GI                NO ERROR APPARENT...
0062                 *
0063                 *                                TABLES (PHYSTB)
0064                 *                                USED BY DRIVER
0065 P0035 0091      TTY      NUM      $91               TTY FUNC CODE
0066 P0036 0041 P FWA         ADC      BUF               CURRENT BUFFER ADDRESS
0067 P0037 0064 P LWA         ADC      BUF+35            LWA+1
0068 P0038 0001      STAT     BZS      STAT(1)           STATUS WORD
0069 P0039 0001               BZS      FLAG(1)
0070                 *
0071                 *                                INTERRUPT RESPONSE
0072                 *                                NOT PART OF DRIVER
0073 P003A E8FA      INTRES   LDQ*     TTY
0074 P003B 02FE               INP      -1                STATUS TTY
0075 P003C 68FB               STA*     STAT              SAVE IT
0076 P003D 0FCD               ALS      13                CK INTERRUPT BIT
0077 P003E 0121               SAP      GI-*-1
0078 P003F 18D3               JMP*     CONT              GO TO TTY CONTINUATOR
0079 P0040 18FF      GI       NUM      $18FF             GHOST INTERRUPT HANG HERE
0080                 *
0081 P0041 2054      BUF      ALF      35, THIS MESSAGE IS WRITTEN IN INTERRUPT
     P0042 4849
     P0043 5320
     P0044 4D45
     P0045 5353
     P0046 4147
     P0047 4520
     P0048 4953
     P0049 2057
     P004A 5249
     P004B 5454
     P004C 454E
     P004D 2049
     P004E 4E20
     P004F 494E
     P0050 5445
     P0051 5252
     P0052 5550
     P0053 5420
     P0054 4D4F
     P0055 4445
     P0056 2020
     P0057 2020
     P0058 2020
     P0059 2020
     P005A 2020
     P005B 2020
```

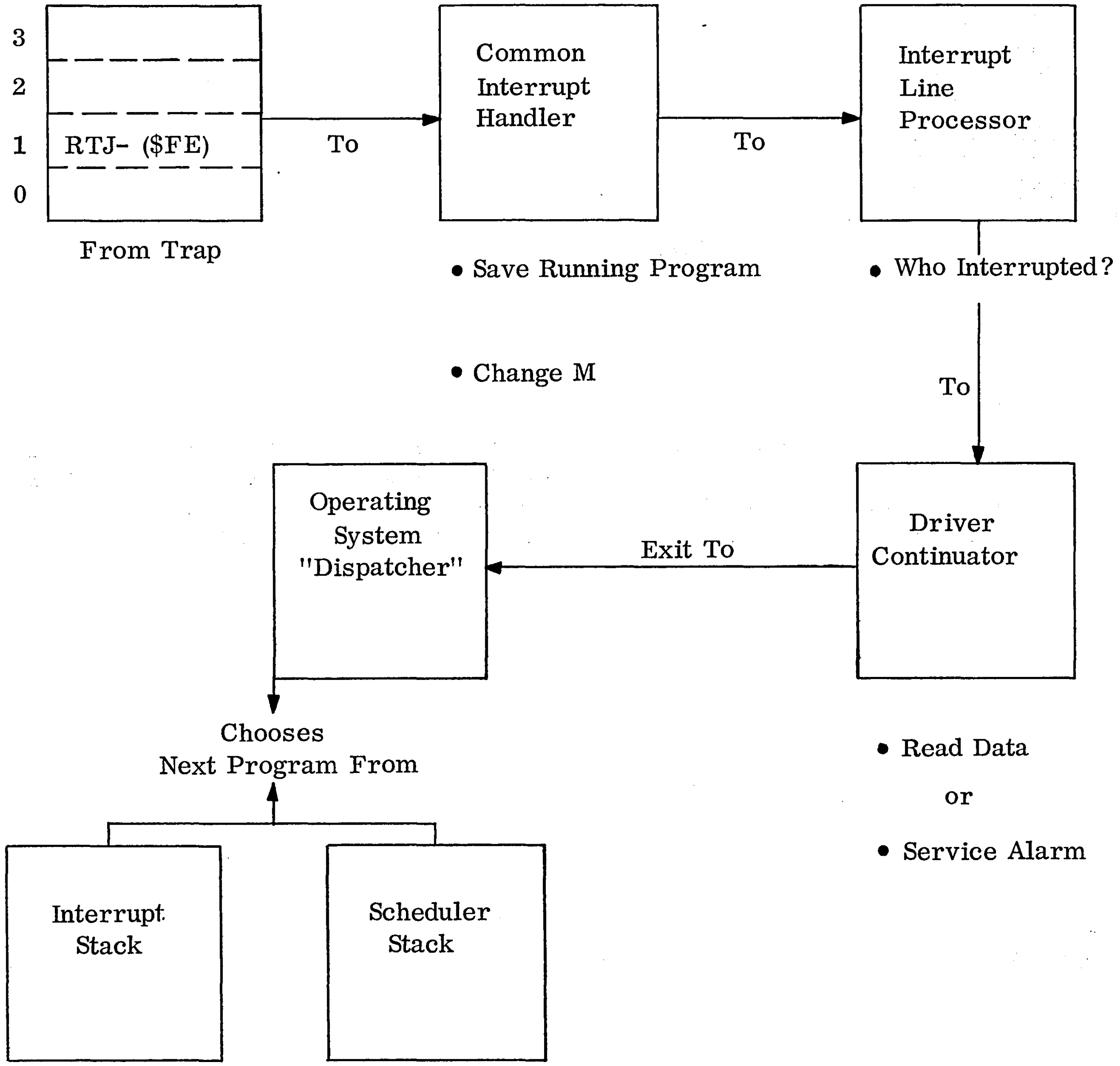
```
        P005C 2020
        P005D 2020
        P005E 2020
        P005F 2020
        P0060 2020
        P0061 2020
        P0062 2020
        P0063 2020
  0082                 *
  0083                                END
```

```
  I       00FF  ADISP   00EA  TTYI    0000P CONT    0013P ERR     002DP
  INTRES  003AP INIT    0005P OK      0017P DATA    001AP LOWER   0022P
  OUTPUT  0024P CMPLET  002CP TTY     0035P FWA     0036P LWA     0037P
  STAT    0038P FLAG    0039P GI      0040P BUF     0041P
```

```
J
*P
J
*ASSEM
OPTIONS  LX
J
*P
J
*L, 8
J
THIS MESSAGE IS WRITTEN IN INTERRUPT MODE
```

13.7

Figure 35. Interrupt Flow



13.8 The following is an example of one of the ADSD bulletins, illustrating interrupt processing routines.

ANALOG-DIGITAL SYSTEMS DIVISION
INFORMATION BULLETIN

CONTROL DATA
CORPORATION

Number: 4

December, 1967

INTERRUPT PROCESSING ROUTINES

This Information Bulletin briefly describes the software involved in processing an interrupt. Although interrupt processing is not new, it is often misunderstood. An Interrupt Service Routine varies in complexity depending on the hardware constraints and user requirements. The Interrupt Service Routine used in the CONTROL DATA 1700 Computer Operating System is not a "closed" routine; rather, it is a group of subroutines which are linked together to provide the flexibility required by today's state-of-the-art programming techniques.

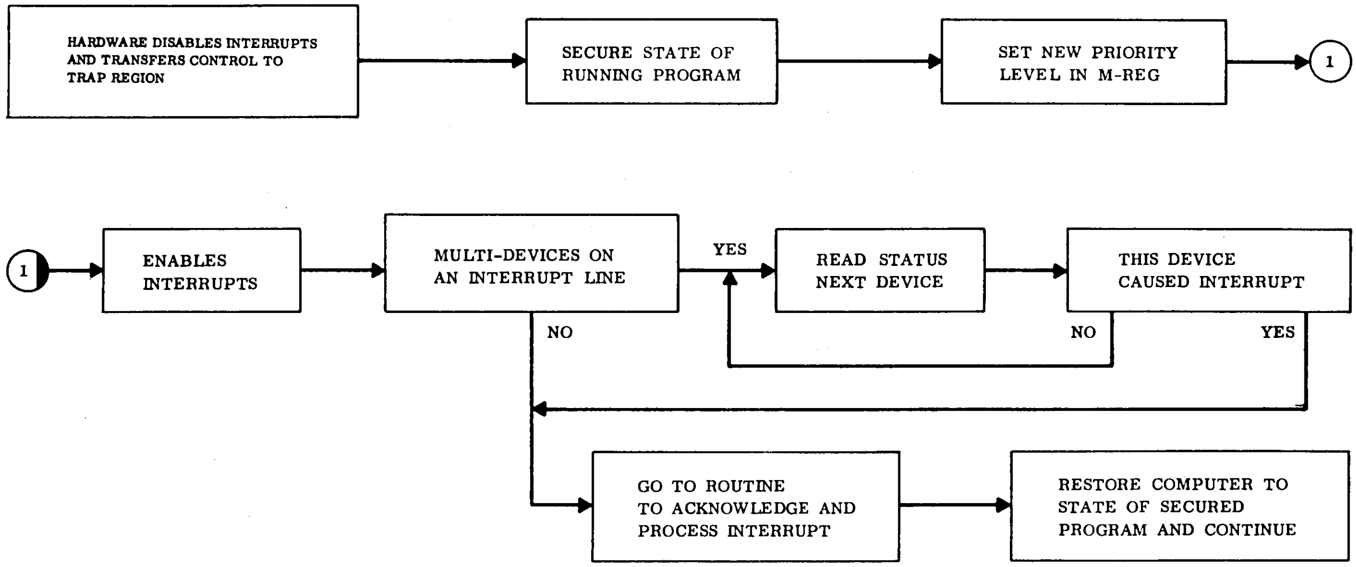
The following conditions must be met in the CDC 1700 Computer before an interrupt can be detected.

- The interrupt system must be enabled.
- An interrupt line must be true.
- The corresponding bit in the interrupt Mask register (M) must be set.

In the following discussion, it has been assumed that the above conditions have been met and that the interrupt is detected or trapped. When an interrupt is trapped, the program sequence is interrupted (or suspended), the address of the instruction to have been executed next is saved, the interrupt system is disabled, and control is transferred to the Interrupt Trap Region.

INTERRUPT TRAP REGION

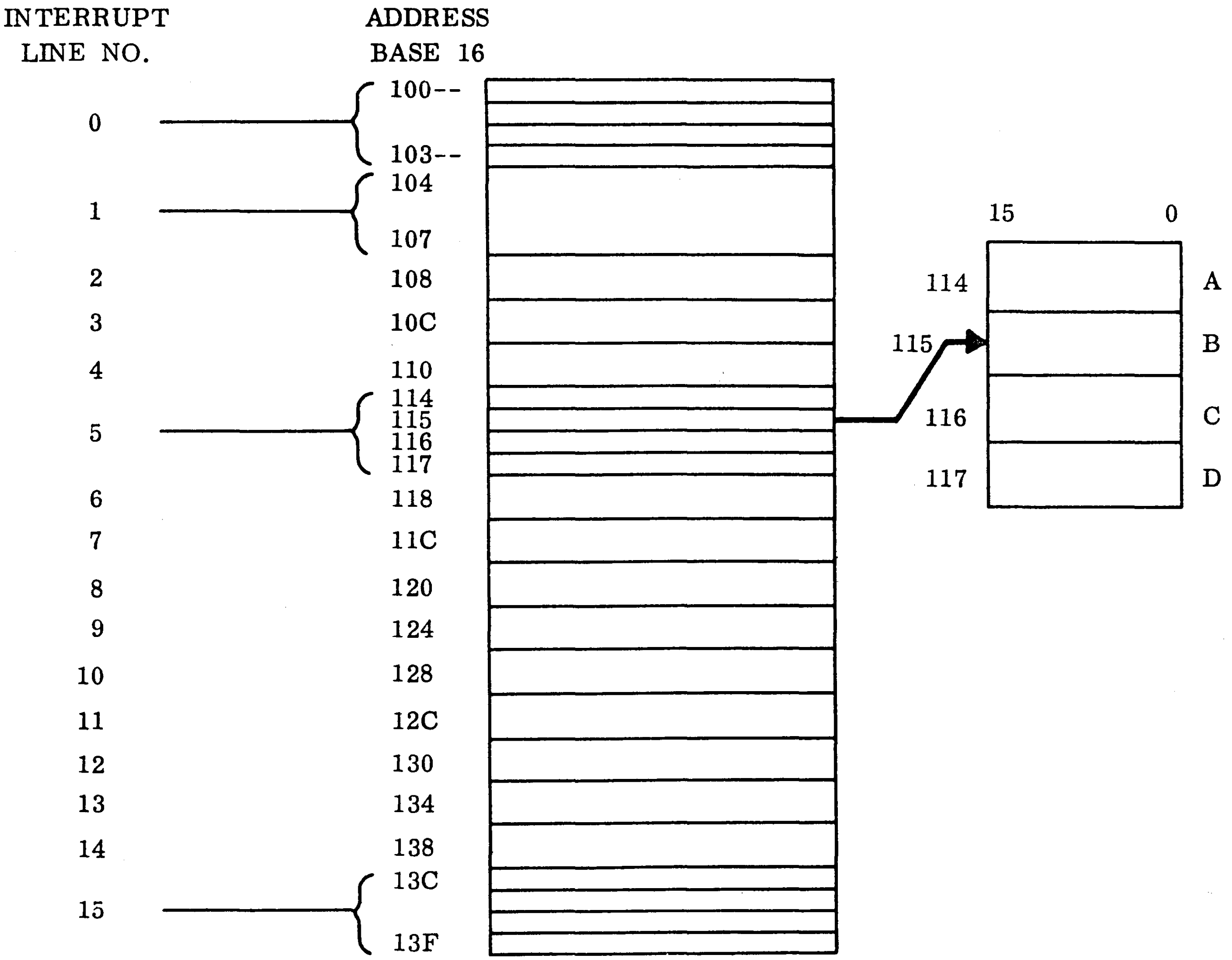
The Interrupt Trap Region is a dedicated area of memory from location 100_{16} through location $13F_{16}$ (64 locations). Four consecutive locations in this region are assigned to each interrupt


HARDWARE DISABLES INTERRUPTS AND TRANSFERS CONTROL TO TRAP REGION
SECURE STATE OF RUNNING PROGRAM
SET NEW PRIORITY LEVEL IN M-REG
1
1
ENABLES INTERRUPTS
MULTI-DEVICES ON AN INTERRUPT LINE
YES
READ STATUS NEXT DEVICE
THIS DEVICE CAUSED INTERRUPT
NO
NO
YES
GO TO ROUTINE TO ACKNOWLEDGE AND PROCESS INTERRUPT
RESTORE COMPUTER TO STATE OF SECURED PROGRAM AND CONTINUE

line. Since there are 16 interrupt lines with 4 memory cells for each line, there are 64 locations assigned to the Interrupt Trap Region. The first four locations are for interrupt line "0," the next four are for interrupt line "1," and so on. When an interrupt is trapped, control is transferred to the second location of the group that is associated with the interrupt line. The exact location is computed as

LOC. $100_{16} + 4_{16}$ (LINE NO.) + 1.

The four memory locations for each interrupt line are shown below followed by an explanation of the contents of each word. (Words are labeled A, B, C, and D.)



A = When an interrupt is trapped, the address of the instruction to have been executed next is saved in A_{14} - A_0. A_{15} will contain the Overflow flip-flop's status when the interrupt was trapped. Loading this memory location as described above is accomplished automatically by the hardware.

B = This location normally contains a one-word indirect Return Jump instruction to a routine that is responsible for preserving the state of the computer's registers (A, Q, I), contents of location "A" and the interrupted programs priority level. In most cases, this Return Jump is to the Common Interrupt Handler. The exact parameters for this word are decided when building the Operating System.

C = This location contains the software priority level associated with this interrupt line. Note that only one priority level can be associated with a given interrupt line; however, any number of interrupt lines, up to 16, can be assigned the same software priority level. The exact parameters for this word are decided when building the Operating System. (See Note 2.)

D = This last location of the four-word group contains the absolute address of the Interrupt Response Routine for this interrupt line.

From the above descriptions it can be seen that the "A" parameter is loaded automatically by the hardware and that control is transferred to the "B" parameter (Return Jump). It is assumed in this discussion that the Return Jump instruction passes control to the Common Interrupt Handler. See Note 1.

COMMON INTERRUPT HANDLER

The Common Interrupt Handler saves the state of the computer (interrupted program) in the Interrupt Stack Region. Information saved includes the contents of the A-, Q-, I-, and P-registers, the Overflow status, and the software priority of the interrupted program. The Common Interrupt Handler then sets the new software priority level, sets the M-register for the new software priority level, enables the interrupt system, and transfers control to the address specified by the "D" parameter, the absolute address of the Interrupt Response Routine.

INTERRUPT RESPONSE ROUTINE

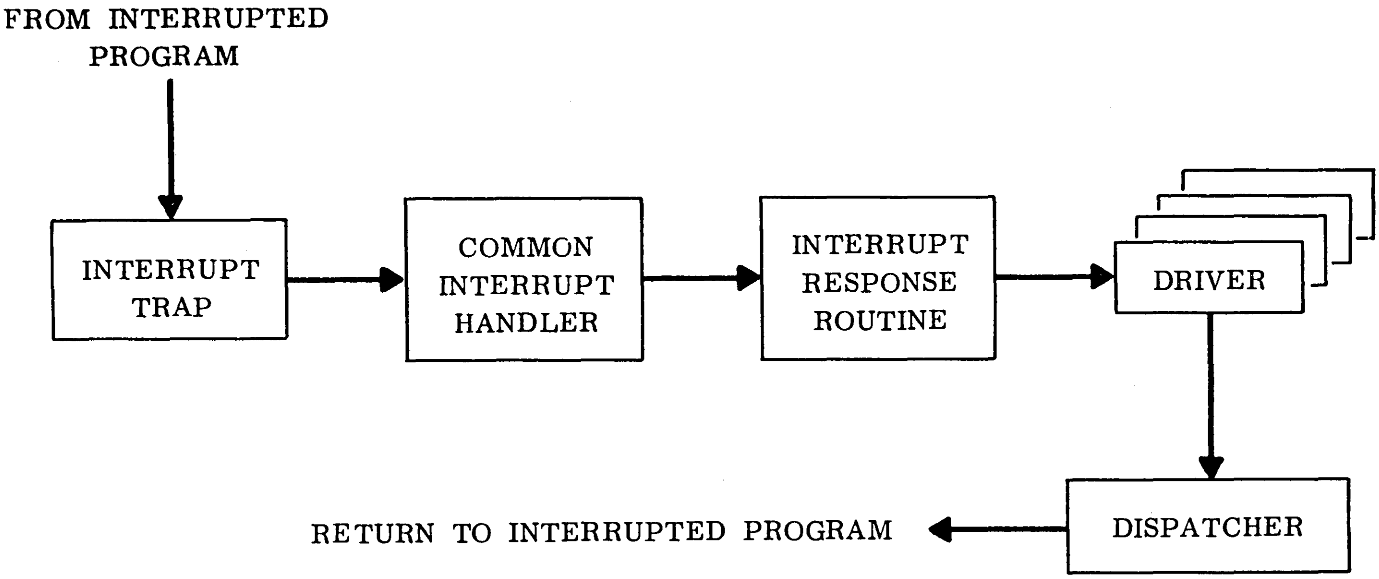
The Interrupt Response Routine is usually a small, user-written subroutine to determine which device caused the interrupt if there is more than one device on the interrupt line and transfer control to the "driver" entry point to process the interrupt.

DRIVERS

The Driver acknowledges (clears) the interrupt line and processes the data as required. Eventually, upon completion of the program initiated by the interrupt, the Driver passes control to a module which is responsible for returning the computer to the original state of the secured program. In the CDC 1700 Computer Monitor, this module is called the Dispatcher.

DISPATCHER

The Dispatcher returns the computer to its original state (its condition at the time of the interrupt) by reloading the A-, Q-, I-, and P-registers, Overflow Status, and priority levels from the Interrupt Stack Region and transfers control back to the address stored as parameter "A" in the Interrupt Trap Region. The M-register is also restored.



SPECIAL NOTES

Note 1

A software rule exists which states that interrupts will not be inhibited for more than 45 memory cycles (50 μs). If an interrupt can be processed in less than 50 μs, the Common Interrupt Handler and the Dispatcher may be bypassed. Therefore, in such cases, parameters "C" and "D" in the Interrupt Trap Region will not be necessary. The one-word indirect Return Jump in location B will transfer control directly to the Interrupt Response Routine or Driver.

Note 2

Interrupt priorities

1. Hardware priorities are from the lower numbered interrupt lines, to the higher numbered interrupt lines. The lower numbered lines being recognized first by the hardware.

2. Software priorities are not necessarily associated with interrupt line numbers, the higher software priority numbers are processed first. For example, a program running at software priority level 5, requests a driver that has a software priority level of 10. The running program would be interrupted and the driver would be initiated (software interrupt).

As an example of how the hardware and software priorities work together assume the following:

1. Processing is being accomplished by the Monitor with interrupts disabled.

2. When the interrupt system is enabled there are two hardware interrupts ready for processing interrupt line-1 (common synchronizer), and software priority level 10, and interrupt line-2 (the timer) software priority level 13.

When the interrupt system is enabled the hardware control will trap the interrupt from line-1 (higher priority, lower line number and disable the interrupt system). This interrupt will be processed through the Common Interrupt Handler requiring about 50 microseconds. The Common Interrupt Handler sets the new software priority level (M-Register Mask), enables interrupts, and transfers control to the Interrupt Response Routine. Interrupt line-2 will trap. This is allowed by the mask setting in the "M" register. The software priority of the "Timer" is always higher than the common synchronizer and must be processed first. However, this could not have been determined prior to the interrupt response routine. Line-2 will continue being processed until completion or until it itself is interrupted.

Therefore, even though the hardware indicated that interrupt line-1 had priority, the software dictated that interrupt line-2 actually had a higher priority.

D. Sandusky

Del Sandusky

13.9 The following is a listing of the interrupt handler.

```
                            NAM   COMMON INTERRUPT HANDLER
                     *      PART NO.     E00610A0010S
                            ENT   ALLIN
       00B8                 EQU   COUNT($B8)
       00EF                 EQU   PRLVL($EF)
       00B7                 EQU   AMASKT($B7)
       0022                 EQU   ZERO($22)
                     *
                     *      AFTER CONTROL IS TRANSFERRED FROM THE INTERRUPT
                     *      TRAP LOCATION TO THE COMMON INTERRUPT HANDLER,
                     *      THE RETURN LOCATION, A, Q AND I REGISTERS AND
                     *      PRIORITY ARE SAVED IN A PUSH-UP POP-DOWN STACK
                     *      BY PRIORITY LEVEL.  THEN THE NEW PRIORITY AND
                     *      MASK ARE SET AND CONTROL IS TRANSFERRED TO THE
                     *      ADDRESS ASSOCIATED WITH THE LINE ON WHICH THE
                     *      INTERRUPT APPEARED.
                     *
       0000                 EQU   XQ(0),XA,(1),XI(2),XR(3),XPL(4),XL(5)
       0001
       0002
       0003
       0004
       0005
P0000  0000          ALLIN  0     0                  LINK TO LEVEL ENTRY
P0001  44BB                 STQ-  (COUNT)          SAVE Q IN STACK
P0002  E0B8                 LDQ-  COUNT            STACK COUNTER AS INDEX
P0003  6201                 STA-  XA,Q                        SAVE A
P0004  C0EF                 LDA-  PRLVL                       SAVE PRIORITY
P0005  6204                 STA-  XPL,Q
P0006  C0FF                 LDA-  I                           SAVE MEMORY
P0007  6202                 STA-  XI,Q                        INDEX REGISTER
P0008  40FF                 STQ-  I                  STACK LOCATION BASE
P0009  0D05                 INQ   XL                          UPDATE STACK
P000A  40B8                 STQ-  COUNT
P000B  E8F4                 LDQ*  ALLIN                       LEVEL LINK
P000C  0DFD                 INQ   -2                 ADJUST TRAP LOCATION
P000D  C622                 LDA-  (ZERO),Q         RETURN LOCATION
P000E  6103                 STA-  XR,I
P000F  40FF                 STQ-  I                  SAVE TRAP LOCATION IN I
P0010  E202                 LDQ-  2,Q                         SET NEW
P0011  40EF                 STQ-  PRLVL                       PRIORITY LEVEL
P0012  C6B7                 LDA-  (AMASKT),Q       SET NEW MASK  I  M  REGISTER
P0013  0400                 EIN
P0014  0821                 TRA   M
P0015  E103                 LDQ-  3,I                JUMP TO PROCESSOR
P0016  1622                 JMP-  (ZERO),Q         LOCATION IN Q
                            END

I      00FF   ALLIN   0000P  COUNT  00BB  PRLVL   00EF  AMASKT   00B7
ZERO   0022   XQ      0000   XA     0001  XI      0002  XR       0003
XPL    0004   XL      0005
```

13.10 The following is a listing of the dispatcher

```
                                NAM   DISPATCHER
                     *          PART NO.     E00610A0020S
                                ENT   DISP,SCHTOP
                                EXT   SCHSTK,SCHLNG
           00B8                 EQU   CONT($B8)
           00EF                 EQU   PRLVL($EF)
           00B7                 EQU   AMASKT($B7)
           0104                 EQU   COMEXT($104)
           0002                 EQU   LPMSK($2),NZERO($12),ZERO($22),ONEBIT($23),ZROBIT($33)
           0012
           0022
           0023
           0033
           0009                 EQU   RCSCHD(9)
           0000                 EQU   XQ(0),XA(1),XI(2),XR(3),XPL(4),XL(5)
           0001
           0002
           0003
           0004
           0005
           00B4                 EQU   TOMPT($B4)
           0001                 EQU   PC(1),PT(2),PQ(3)
           0002
           0003
                     *          UPON COMPLETION OF A PROGRAM, THE DISPATCHER
                     *            DETERMINES THE PROGRAM OF HIGHEST PRIORITY
                     *                WAITING FOR EXECUTION.  IT MAY EITHER BE IN THE
                     *                INTERRUPT STACK OR THE SCHEDULER STACK.
                     *
                     *
     P0000 FFFF      SCHTOP     NUM   $FFFF                  SCHEDULE STACK TOP
     P0001 E0B8      DISP       LDQ-  CONT
     P0002 0DFA                 INQ   -XL                                ADJUST STACK
     P0003 0500                 IIN   0
     P0004 C8FB                 LDA*  SCHTOP                   IF SCHEDLER STACK IS
     P0005 0900                 INA   0
     P0006 0106                 SAZ   RESINT-*-1               EMPTY, CHECK INT. STACK
     P0007 CCF8                 LDA*  (SCHTOP)                 LOAD FIRST WORD
     P0008 A006                 AND-  LPMSK+4                  ISOLATE PRIORITY
     P0009 0821                 TRA   M                        SAVE TEMP. IN  M
     P000A 9204                 SUB-  XPL,Q                  PRIORITY OF HIGHEST INT.
     P000B 0101                 SAZ   RESINT-*-1             GO TO INTERRUPT STACK
     P000C 012C                 SAP   SCHSTC-*-1               GO TO SCHEDLE STACK
                     *
                     *     HIGHEST PROGRAM IS IN THE INTERRUPT STACK.
                     *
     P000D C203      RESINT     LDA-  XR,Q                               SET RETRN LOCATION
     P000E 6C2B                 STA*  (ACOMEX)
     P000F C202                 LDA-  XI,Q                               RESTORE  I
     P0010 60FF                 STA-  I
     P0011 C201                 LDA-  XA,Q                               RESTORE  A
     P0012 40B8                 STQ-  CONT                   STORE INT. STACK BASE
     P0013 E204                 LDQ-  XPL,Q                    RESTORE PRIORITY LEVEL
     P0014 40EF
```

```
046.  P0015 E6B7                LDQ-  (AMASKT),Q         RESTORE MASK
047.  P0016 0811                TRQ   M
048.  P0017 E4B8                LDQ-  (CONT)            RESTORE Q
049.  P0018 0E04                EXI   COMEXT-256
050.                 *


051.                 *     HIGHEST PROGRAM IS IN THE SCHEDLER THREAD.
052.                 *
053.  P0019 080A     SCHSTC     TRM   Q                  PRIORITY TO Q
054.  P001A 40EF                STQ-  PRLVL              SET NEW PRIORITY AND MASK
055.  P001B C6B7                LDA-  (AMASKT),Q
056.  P001C 0821                TRA   M
057.  P001D E8E2                LDQ*  SCHTOP             STORE NEW POINTER
058.  P001E C202                LDA-  PT,Q               TOP OF SCHEDLER THREAD
059.  P001F 68E0                STA*  SCHTOP
060.  P0020 0814                TRQ   A                 TEST IF PRIMARY SCHEDULER
061.  P0021 9819                SUB*  ASCHD             CALL WAS MADE.
062.  P0022 0138                SAM   SCHSEC-*-1
063.  P0023 9818                SUB*  ASCLNG
064.  P0024 0126                SAP   SCHSEC-*-1
065.  P0025 C0B4                LDA-  TOMPT              IF PRIMARY CALL RELEASE
066.  P0026 6202                STA-  PT,Q               STACK POSITION AND PLACE
067.  P0027 40B4                STQ-  TOMPT             ON EMPTY THREAD.
068.  P0028 C201                LDA-  PC,Q               LOAD ABSOLUTE ADDRESS
069.  P0029 6C10                STA*  (ACOMEX)          STORE INTO COMEXT
070.  P002A 180C                JMP*  SCHXIT
071.  P002B C622     SCHSEC     LDA-  (ZERO),Q           TEST IF ABSOLTE OR RELATIVE
072.  P002C A02B                AND-  ONEBIT+8
073.  P002D 0101                SAZ   SCH1-*-1          CALL.  SKIP IF ABSOLUTE
074.  P002E 0814                TRQ   A                  ADDRESS 1ST WD OF CALL
075.  P002F A011     SCH1       AND-  LPMASK+15
076.  P0030 8032                ADD-  ONEBIT+15
077.  P0031 8201                ADD-  PC,Q               ADD REL.  ADDRESS OR IF
078.  P0032 A011                AND-  LPMSK+15
079.  P0033 6C06                STA*  (ACOMEX)           A=0, ABS ADDRESS AND STORE
080.  P0034 0844                CLR   A                  ZERO INTO THREAD
081.  P0035 6202                STA-  PT,Q               COMPLETION INDICATION
082.  P0036 0814     SCHXIT     TRQ   A                  PASS POINTER TO CALL IN A
083.  P0037 E203                LDQ-  PQ,Q                               PASS,Q
084.  P0038 0E04                EXI   COMEXT-256
085.  P0039 0104     ACOMEX     ADC   COMEXT
086.  P003A 7FFF X   ASCHD      ADC   SCHSTK            SCHED. STACK LOCATION
087.  P003B 7FFF X   ASCLNG     ADC   SCHLNG            SCHED. STACK LENGTH LOC.
088.                            END

I        00FF    DISP     0001P   SCHTOP   0000P   CONT     00B8    PRLVL    00EF
AMASKT   00B7    COMEXT   0104    LPMSK    0002    NZERO    0012    ZERO     0022
ONEBIT   0023    ZROBIT   0033    RCSCHD   0009    XQ       0000    XA       0001
XI       0002    XR       0003    XPL      0004    XL       0005    TOMPT    00B4
         0001    PT       0002    PQ       0003    RESINT   000DP   SCHSTC   0019P
SCHSEC   002BP   SCH1     002FP   SCHXIT   0036P   ACOMEX   0039P   ASCHD    003AP
ASCLNG   003BP   SCHLNG   003BX   SCHSTK   003AX
```

CHAPTER XIV

LIBEDT EXAMPLES

14

CHAPTER XIV - LIBEDT Examples

14.0 INTRODUCTION

The MSOS library editing program, LIBEDT, can be used effectively for adding routines to the program library, exchanging old routines in the program or system library for new ones, and for many utility functions such as transferring records from one logical unit to another, absolutizing programs, etc.

The LIBEDT chapter of the MSOS reference manual describes the features of LIBEDT very accurately. Therefore, this chapter will consist simply of examples using LIBEDT.

14.1 MASS MEMORY REPLACE

The first example, teletype printout, shows a sample of the checking which could be done while replacing a mass memory module in the system library. The replacement module is larger than the original one. The comments on the listing are self-explanatory.

LIBEDT MASS MEMORY REPLACE

```
J
*LIBEDT
  LIB

  IN

*DL                      Dump program library
  IN

*DM                      Dump system library
  IN

~~*M,26,,M~~    <------  Cancel this statement with rub-out, LF, CR
*K,12           <------  Input on lun 2 PTR
  IN

*M,26,,M        <------  Replace MM ordinal #26 in system library
L,02  FAILED  02
ACTION
RP
L,02  FAILED  02         Add and link subroutines from PTR
ACTION
RP
L,02,  FAILED  02
ACTION
CU

E *E                     Patch missing externals from core resident entry points
MI
*Z
J
*P                       Following checking is only to illustrate debugging features:
J
*LIBEDT
  LIB

  IN

*DM             <------  Check to see if ordinal changed (sector address should be larger because new
  IN                     program is larger than one it replaces; therefore, it will be put in first avail-
                         able scratch sector)
*Z
J
*P
J
*L,8
J
*SR                      Dummy L and X to get sys. rec. package
J
*X,,
RE
*M2E6

RE
*M2E6,0,2F1              Dump sector $2E6 (on LP) to check program

RE
*M2F1,0,2F5

RE
*DC0,C1         <------  Dump core $C0 and $C1 - scr sector address

RE
```

14.2 GTFILE REQUEST FOR SYSTEM INITIALIZER

This example is not actually a LIBEDT example. However, it is included because it pertains to the system initializer. GETSI is a program which can be modified to be included in any program library (in relocatable binary form); it can then be called into execution in the background by:

```
J
*P
J
*SI
```

The program does a GTFILE request to bring in a file, SYSINI, from the program library to address $6000. SYSINI is actually an absolute copy of the system initializer. GETSI then jumps to $6000 to execute the system initializer. This is very handy because it provides for storing the system initializer on the program library so it can be called in without having to key in a bootstrap loader from the console.

```
0001                          NAM      GETSI
0002                          ENT      SI
0003  P0000 5359    FILNAM    ALF      *,SYSINI* ◄——— Name of FILE in program library
      P0001 5349
      P0002 4E49
0004  P0003 5349    BUF       ALF      *,SI   IN* ◄——— MSG Buffer for TTY
      P0004 2049
      P0005 4E20
0005        6000    SIADDR    EQU      SIADDR($6000)      File to go at $6000
0006  P0006 0000    SI        0        0
0007                GETFIL    GTFILE   GOT,(FILNAM),SIADDR,,,0,0,1
0007  P0007 54F4                                                            CP = 1
0007  P0008 1A01          Completion address                              RP = 0
0007  P0009 0014 P     after SYSINI is brought    Address               X bit = 0 (not blank)
      P000A 0000                    into core     where       Core
0007  P000B 08C2                                  file name   address     Disk address
      P000C 0000                                  is          where       left blank;
      P000D 6000                                              file is     program li-
0007  P000E 0000                                              to go       brary will be
      P000F 8000 P                                                        searched
0008                          EXIT (wait for completion)
0008  P0010 54F4
0008  P0011 0A00
0009                          EXIT         (unnecessary)
0009  P0012 54F4
0009  P0013 0A00
0010  P0014 017B    GOT       SQM      NOGOOD
0011                SIIN      FWRITE   $FC,WROTE,BUF,3,A,0,1,I,0
0011  P0015 54F4                                                  X bit = 0 (not blank)
0011  P0016 0C01                   LUN                           Indirect bit referring to $FC
0011  P0017 001E P         (std comment     COMPL                CP = 1
      P0018 0000                device)     ADDR    MSG        RP = 0
0011  P0019 18FC                                    BUF    ASCII
0011  P001A 0003                                        3 words
      P001B 0003 P
0012                          EXIT
0012  P001C 54F4              Wait until Write is done
0012  P001D 0A00
0013  P001E 0171    WROTE     SQM      NOGOOD
0014  P001F 1CED              JMP*     (GETFIL+6) ◄——— Jump to beginning address of
0015                NOGOOD    EXIT                     SYSINI which is $6000
0015  P0020 54F4
0015  P0021 0A00
0016                          END      SI
```

This program may be reassembled for any system. Change EQU for the desired high core address where the system initializer should run when in core. The system initializer should be stored in the program library under the file name SYSINI, as an absolute file.

14.3 ADDING PROGRAMS AND FILES TO THE PROGRAM LIBRARY

The following example shows the use of LIBEDT to absolutize the system initializer and put it in the program library under file name SYSINI. Then GETSI is put in the program library as a relocatable binary program.

A subsequent system initialization run was made to check out GETSI; the listing shows it beginning to execute.

Put System Initializer in Program Library

```
MI
*P
J
*LIBEDT      <-----------------  Call in LIBEDT
   LIB

   IN
                                 Input on lun 2 PTR - sys. ini. in rel. bin. form
*K,I2,P8     <-----------------  Absolutize on lun 8 (disk scratch)
   IN

~~P,F~~
   ~~L04~~
   IN

*P,F          -----------------  Absolutize in 96-word blocks for disk
L,16  FAILED  01
ACTION                           Printer failed
RP
L,02  FAILED  02
ACTION                           Out of tape.  Complete absolutizing
CU

E *          <-----------------  Unpatched externals (unnecessary modules were taken out of sys. ini.) - ignore
   IN

*K,I8        <-----------------  Input now on 8
   IN

~~*N~~
~~:N,~~
  ~~L04~~
   IN

*N,SYSINI,,,B  <---------------  File SYSINI, binary, placed in pgm. lib.
   IN

*K,I2        <-----------------  Input now on lun 2 PTR (GETSI)
   IN

*L,GETSI     <-----------------  Put GETSI in pgm. lib. in rel. bin. form
  L08
   IN

*Z           <-----------------  Sign off LIBEDT
J
                                        CHECK IT OUT
*P
J            <-----------------  Turn off protect switch here!
*SI          <-----------------  Call in GETSI
J

SI  IN       <-----------------  GETSI gets SYSINI and types SI  IN
SI           <-----------------  System initializer types SI
*I,1         <-----------------  Begin initialization!

Q
*V
*Y,QQTEST,2,QQQ8AB,3,QQSPAC,4
*YM,LOADSD,1,JOBENT,2,JOBPRO,3,JPLOAD,4
*YM,JPST,5,JPCHGE,6,JBKILL,7,JPT13,8
*YM,RCOVER,9,LIBEDT,10,MOD1,11,MOD2,12
*YM,MOD3,13,BRKPT,14,RESTOR,15,MOD4,16
*YM,DEBUG,17,DSUTIL,18,TYPEID,19,DSTART,20,RSTART,21
*YM,QQFMT1,22,QQCOM,23,QQANAB,24,QQUTL1,25        etc.
```

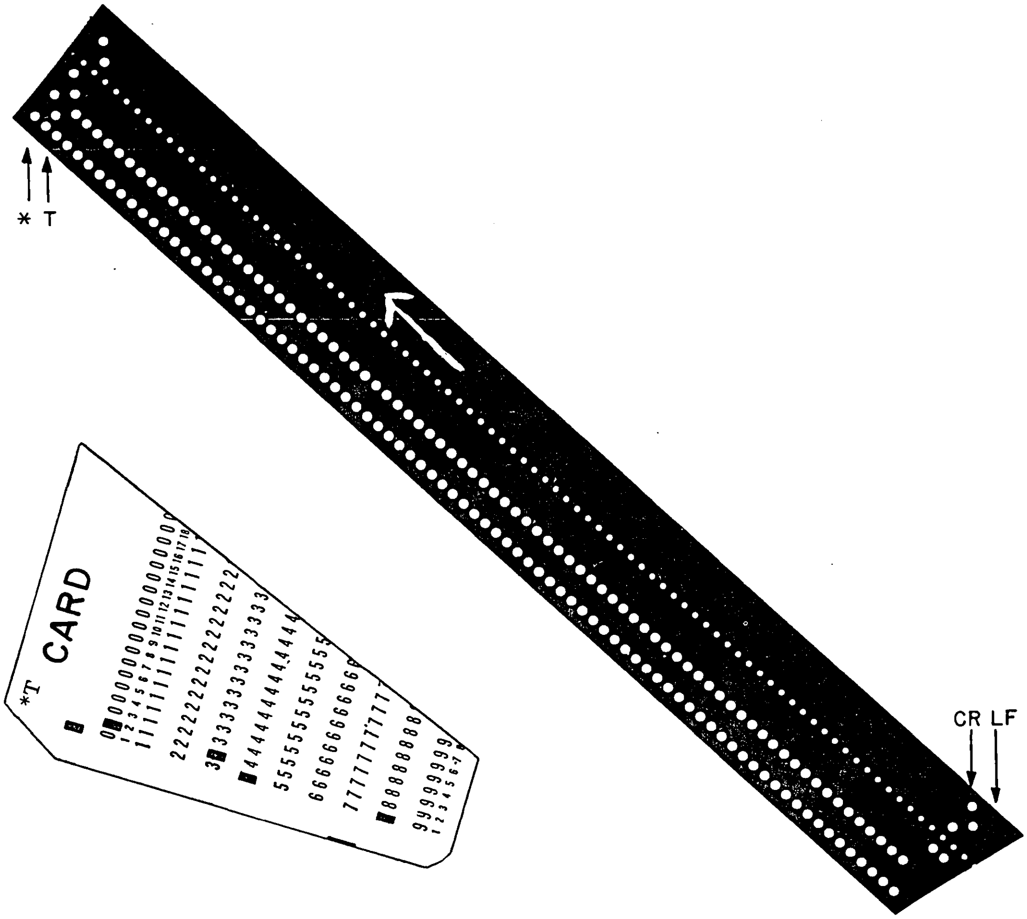
14.4 TRANSFERRING RECORDS

In this example, LIBEDT was used to transfer a card image from the card reader to paper tape. The *T image in the example was then attached to the end of a series of paper tape programs being loaded, to cause the loader to end loading.

TRANSFER RECORDS FROM ONE LUN TO ANOTHER

LIBEDT *T (*T option is used to transfer rel. bin. paper tapes which are SI input to high portion of disk so that disk input may be made to SI.)

```
*Z
J
*P
J
*LIBEDT
   LIB

   IN


*T,12,A,11,A,1
   IN                    Transfer from LUN 12 (CR), ASCII mode, to LUN 11 (PTP),
                         ASCII mode, one record (1 card)
```



14.5 ABSOLUTIZING AND LINKING SUBPROGRAMS

Next, LIBEDT was used to build a utopia system. This involves loading, absolutizing, and linking a series of relocatable binary paper tapes. Only the routines applicable to the particular configuration were included. The new utopia in core can then be used to punch out an absolute paper tape image of itself.

14.5 BUILD UTOPIA SYSTEM USING LIBEDT

```
*P
J
*LIBEDT  ◄———————— Call in LIBEDT
  LIB

  IN

*K,I2,P11 ◄———————— Assign input LUN 2 (PTR) output LUN 11 (PTP)
  IN

*P  ◄———————————— Load and absolutize tapes
L,02 FAILED 02 }
ACTION         }    UTOPIA
RP             }
L,02 FAILED 02 }    UMAINF
ACTION         }
RP             }
L,02 FAILED 02 }    USILLY
ACTION         }
RP             }
L,02 FAILED 02 }    U1711
ACTION         }
RP             }
L,02 FAILED 02 }    UDISK
ACTION         }
RP             }
L,02 FAILED 02 }    CARDIN
ACTION         }
RP             }
L,02 FAILED 02 }    U1729
ACTION         }
RP             }
L,02 FAILED 02 }    U601MT
ACTION         }
RP             }
L,02 FAILED 02 }    UTODMP
ACTION         }
RP             }
L,02 FAILED 02 }    UDALP
ACTION         }
RP             }
L,02 FAILED 02 }    CKOUT
ACTION         }
RP             }
L,02 FAILED 02 }    UTLAST
ACTION         }
CU    ◄———————— Wind up load and link

E *   ◄———————— Externals missing; ignore; punch tape
  IN

*Z    ◄———————— Sign off LIBEDT
J
```

BUILD UTOPIA PRINT OUT

```
          UTOPIA    2210
          UMAINF    2392
          USILLY    25A8
          U1711     3258
          UDISK     343E
          CARDIN    3680
          U1729     3707
          U601MT    38A9
          UTODMP    39BE
          UDALP     3AA2
          CKOUT     444A
          UTLAST    4F23
     E10
     CKD
     PCR
     DKM
     MAR
     VDM
     LPRINT
     RDM
     WDM
```

Routine names and addresses where loaded

Missing Externals
(These are for modules not included in this system)

Applicable Utopia routines for customer's configuration can be loaded and linked together, then an absolutized tape can be punched. It can then be loaded by a cksum loader (which must be loaded by a bootstrap loader).

Utopia modules could also be loaded and linked and put on program library in rel bin form (with Libedt *L). Then Utopia could be called from TTY with *UTOPIA.

APPENDIXES

A

APPENDIXES
TABLE OF CONTENTS

APPENDIX A

APPENDIX A

Ref. No.	Reference Title	Publication No.
1	Reference Manual	60153100
2	Codes	60163500
3	Assembler Reference Manual	60171600
4	1700 Macro/Assembler Reference Manual	60176300
5	Utility Reference Manual*	60172300
6	Operating System Reference Manual	60174600
7	1700 4K Assembly System (ADB)	60176500
8	Input/Output Specifications Manual	60165800
9	Systems Manual	60152900
10	ADSD General Information Manual	60187600

*Utility Assembler requires 8K system.

APPENDIX B

APPENDIX B

1700 INSTRUCTION EXECUTION TIMES

STORAGE REFERENCE

	Instruction	Execution Time (microseconds)*
LDA	Load A	2.2
STA	Store A	2.2
LDQ	Load Q	2.2
STQ	Store Q	2.2
ADD	Add A	2.2
SUB	Subtract	2.2
ADQ	Add Q	2.2
AND	AND with A	2.2
EOR	Exclusive OR with A	2.2
RAO	Replace Add One in Storage	3.3
MUI	Multiply Integer	7.0
JMP	Jump	1.1
RTJ	Return Jump	2.2
DVI	Divide Integer	9.0
SPA	Store A, Parity to A	2.2

*Add 1.1 microsecond if Storage Index Register is used.
Add 1.1 microsecond for each level of Indirect Addressing.
Add 1.1 microsecond for two word instructions.

REGISTER REFERENCE

	Instruction	Execution Time (microseconds)
SLS	Selective Stop	1.1
INP	Input to A	1.1 min., 10 max.
OUT	Output from A	1.1 min., 10 max.
ENA	Enter A	1.1
ENQ	Enter Q	1.1
INA	Increase A	1.1
INQ	Increase Q	1.1
ARS	A Right Shift	1.1 +(shift count *.1)
QRS	Q Right Shift	
ALS	A Left Shift	
QLS	Q Left Shift	

	Instruction	Execution Time (microseconds)
LRS	Long Right Shift	1.1 + (shift count *.2)
LLS	Long Left Shift	
NOP	No Operation	1.1
EIN	Enable Interrupt	1.1
IIN	Inhibit Interrupt	1.1
EXI	Exit Interrupt State	2.2
SPB	Set Program Protect	2.2
CPB	Clear Program Protect	2.2

INTERREGISTER

	Instruction	Execution Time (microseconds)
SET	Set to Ones	1.1
CLR	Clear to Zero	
TRA	Transfer A	
TRM	Transfer M	
TRQ	Transfer Q	
TRB	Transfer Q V M	
TCA	Transfer Complement A	
TCM	Transfer Complement M	
TCQ	Transfer Complement Q	
TCB	Transfer Complement Q V M	
AAM	Transfer Arithmetic Sum A, M	
AAQ	Transfer Arithmetic Sum A, Q	
AAB	Transfer Arithmetic Sum A, Q V M	
EAM	Transfer Exclusive or A, M	
EAQ	Transfer Exclusive or A, Q	
EAB	Transfer Exclusive or A, Q V M	
LAM	Transfer Logical Product, A, M	
LAQ	Transfer Logical Product A, Q	
LAB	Transfer Logical Product A, Q V M	
CAM	Transfer Complement Logical Product A, M	
CAQ	Transfer Complement Logical Product A, Q	
CAB	Transfer Complement Logical Product A, Q V M	

SKIPS

	Instruction	Execution Time (microseconds)
SAZ	Skip if A = +0	1.1
SAN	Skip if A ≠ +0	
SAP	Skip if A = +	
SAM	Skip if A = -	
SQZ	Skip if Q = +0	
SQN	Skip if Q ≠ +0	
SQP	Skip if Q = +	
SQM	Skip if Q = -	
SWS	Skip if Switch Set	
SWN	Skip if Switch Not Set	
SOV	Skip on Overflow	
SNO	Skip on No Overflow	
SPE	Skip on Storage Parity Error	
SNP	Skip on No Storage Parity Error	
SPF	Skip on Program Protect Fault	
SNF	Skip on No Program Protect Fault	

APPENDIX C

APPENDIX C

UTILITY ASSEMBLER PSEUDO INSTRUCTIONS

NAM n

First instruction in source program. The name, n, if given, identifies the program. A numeric location field for this instruction specifies the absolute starting location for the program; a symbolic location field is ignored.

END e

Last instruction in source program. The entry point, e, if given, is the start of the program.

ENT n_1, n_2 , . . .

Entry points, n_i, that may be referred to by other programs.

EXT n_1, n_2 , . . .

External locations, n_i, of other programs that are referred to by this program.

EXT*n_1, n_2 , . . .

Similar to EXT except that references to the external names, n_i, are made relative.

BSS $n_1(s_1), n_2(s_2)$ . . .

Reserves a block of program storage locations. The names, n_i, identify segments of s_i words in length.

BZS $n_1(s_1), n_2(s_2)$, . . .

Similar to BSS. In addition, zero fills the block.

COM $n_1(s_1), n_2(s_2)$ . . .

Reserves a block of common storage locations. The names, n_i, identify segments of s_i words in length. Data may not be prestored in the common block.

DAT $n_1(s_1), n_2(s_2)$, . . .

Reserves a block of data storage locations. The names, n_i, identify segments of s_i words in length. Data may be prestored in the data block.

ADC e_1, e_2 , . . .

Defines address expressions to be stored as address constants. The addresses may be positive or negative and absolute or relocatable. Parentheses indicate indirect addressing.

ALF n <2n characters>

Stores ASCII alphanumeric characters into consecutive locations of program storage.

NUM c_1, c_2 , . . .

Stores decimal or hexadecimal constants, c_i, in consecutive locations in program storage.

UTILITY ASSEMBLER PSEUDO INSTRUCTIONS (CONT)

ORG e
Assemble subsequent instructions beginning at the address expression, e, which may be program relocatable, data relocatable or absolute.

ORG* e
Resumes assembling instructions immediately after the location preceding the ORG instruction or the first ORG if more than one in a string.

EQU $n_1(e_1), n_2(e_2)$,...
Equates names, n_i, to address expression, e_i.

NLS
Inhibits list output of assembly.

LST
Enables list output of assembly following issuance of a NLS instruction. (Listing is automatic unless NLS is given.)

SPC v
Spaces v lines on the typewriter.

APPENDIX D

APPENDIX D

ASSEMBLY ERROR MESSAGES

ERROR LISTING

A list of errors occurring in passes 1 and 2 precedes the program listing on the standard comment I/O unit. If the L option is selected, errors in pass 3 precede the source line on the list output. A decimal error count is printed at the end of each subprogram. If L is not selected, error messages are output on the standard comment unit.

Format for pass 1 and 2 error messages:

Column	Contents
1-3	3-digit line number
4	space
5-6	**
7-8	2-character error code

Format for pass 3 error messages:

Column	Contents
1-6	******
7-8	2-character error code
9-18	**********

Following are the error message codes and their definitions.

Message	Meaning
**DS	Doubly defined symbol. A name in 1) location field of a machine instruction or an ALF, NUM, or ADC pseudo instruction or 2) address field of an EQU, COM, DAT, EXT, BSS, or BZS pseudo instruction has been used again in one of the above fields.
**UD	Undefined symbol in an address expression.
**SO	Available storage for saving symbol names exceeded; no more names may be defined. Symbol table overflow.

Message   Meaning

**EX   Illegal expression. One of the following:

1) No forward referencing of some symbolic operands

2) No relocation of certain expression values

3) A violation of relocation

4) Possibly a comment is being interpreted as an address field in an instruction which has no address field

**LB   Numeric or symbolic label contains illegal character. The label is ignored.

**IX   Illegal index register; specified by symbol other than Q, I or B.

**OP   1) Illegal symbol in operation code field, e.g., LDI TAG
2) Illegal operation code terminator
3) Could also be caused by error in macro

**OR   Numeric or symbolic operand in address expression contains illegal characters.

**RG   Illegal register for interregister instructions, e.g., CLR I

1) Symbol other than A, Q, or Mused in address field of interregister instruction, or the same symbol used more than once.

2) Registers separated by other than a comma.

**RL   1) Violation of relocation.
2) Violation of a rule for instructions which require the expression value to

a) be absolute
b) have no forward referencing of symbolic operands.

**OV   Numeric operand overflow-numeric value greater than allowed, e.g., 1 wd. rel. more than $7F.

**SQ   Sequence error

**MD   Macro definition error

**MC   Macro instruction error

**PP   Error in previous pass of compilation

**NN   No NAM statement. Blank name will be inserted by assembler.

**MO   Overflow of the load-and-go area of mass storage.

LOADER ERROR MESSAGES

All loading error messages appear on the standard print output device.

E01 Irrecoverable input error; terminates loading.

E03 Illegal or out-of-order input block; terminates load. Also, this diagnostic appears on the comment device when illegal input from that device is detected. The comment device is interrogated for a new statement.

E04 Incorrect common block storage reservation. Occurs if the first NAM block to declare common storage does not declare the largest amount. The loader uses the previously declared length and continues.

E05 Program too long or loader table overflow. Terminates loading. Occurs if program to be loaded exceeds available unprotected core. It may be possible to load the program by rearranging the order of loading to insure entry points are defined before they are referred to as external symbols. Loader produces a memory map and list of unpatched externals prior to terminating the load.

E06 Attempt to load information in protected core; terminates loading.

E07 Attempt to ORG into part of data storage beyond assigned block; terminates loading.

E08 Duplicate entry point; loading continues. The succeeding program is loaded, overlaying the program with the duplicate entry point.

E10 Unpatched external. External name is printed following diagnostic. The operator may choose to terminate the job or continue execution in spite of unpatched externals (with *(CR)). A *E(CR) will cause the loader to search the directory of core resident entry points for the missing external.

E11 Error in HEX block; loader skips remainder of block and resumes loading with the next block. Image of HEX block in error is printed following diagnostic.

E12 Two programs reference same external name, one with absolute addressing, the other relative addressing.

E13 Undefined or missing transfer address; occurs when loader does not encounter a name for the transfer address to begin execution, or the name encountered is not defined in loader's table as an entry point name.

JOB PROCESSOR ERROR MESSAGES

PARITY, hhhh — Memory parity error at location $hhhh_{16}$. Message appears on output comment device.

OV — Overflow of volatile storage. Message appears on output comment device.

L,nn FAILED ee — Informs operator of device failure.

nn logical unit number

ee code indicating cause of failure as follows:

00	I/O hangup
01	Internal or external reject
02	Alarm
03	Parity error
04	Checksum error
05	Internal reject
06	External reject

ALT,aa — Informs operator an alternate device, aa, has been assigned.

ACTION — Requests operator action when a failed device has no alternate. The device is identified in the FAILED diagnostic.

J01,hhhh — Program protect violation. hhhh is current contents of P register. Standard print output device.

J02,hhhh — Illegal request or parameters at location $hhhh_{16}$. Standard print output device.

J03,statement — Unintelligible control statement is output with the diagnostic. Standard output device.

J04,statement — Illegal or unintelligible parameters in control statement. Standard print output device.

J05 — Statement entered after manual interrupt illegal. Output comment device.

J06,hhhh — A threadable request was made at level one when noprotect processor stack space was available, or an unprotected threaded request was made at level one. Standard print output device.

J07,hhhh — Unprotected program tried to access protected device from location hhhh. Standard print output device.

J08,hhhh — Attempt to access read only unit for write, or write only unit for read, or an attempt to access an unprotected request on a protected unit.

APPENDIX E

APPENDIX E

AMERICAN STANDARD CODE FOR INFORMATION INTERCHANGE (ASCII)

Bit Configuration	Symbol	Bit Configuration	Symbol
000 0000	NUL	001 1011	ESC
000 0001	SOH	001 1100	FS
000 0010	STX	001 1101	GS
000 0011	ETX	001 1110	RS
000 0100	EOT	001 1111	US
000 0101	ENQ	010 0000	SP
000 0110	ACK	010 0001	!
000 0111	BEL	010 0010	"
000 1000	BS	010 0011	#
000 1001	HT	010 0100	$
000 1010	LF	010 0101	%
000 1011	VT	010 0110	&
000 1100	FF	010 0111	'
000 1101	CR	010 1000	(
000 1110	SO	010 1001	)
000 1111	SI	010 1010	*
001 0000	DLE	010 1011	+
001 0001	DC1	010 1100	,
001 0010	DC2	010 1101	-
001 0011	DC3	010 1110	.
001 0100	DC4	010 1111	/
001 0101	NAK	011 0000	0
001 0110	SYN	011 0001	1
001 0111	ETB	011 0010	2
001 1000	CAN	011 0011	3
001 1001	EM	011 0100	4
001 1010	SS	011 0101	5

ASCII (Cont)

Bit Configuration	Symbol	Bit Configuration	Symbol
011 0110	6	101 0010	R
011 0111	7	101 0011	S
011 1000	8	101 0100	T
011 1001	9	101 0101	U
011 1010	:	101 0110	V
011 1011	;	101 0111	W
011 1100	<	101 1000	X
011 1101	=	101 1001	Y
011 1110	>	101 1010	Z
011 1111	?	101 1011	[
100 0000	\	101 1100	~
100 0001	A	101 1101	]
100 0010	B	101 1110	Λ
100 0011	C	101 1111	-
100 0100	D	110 0000	@
100 0101	E	110 0001	a
100 0110	F	110 0010	b
100 0111	G	110 0011	c
100 1000	H	110 0100	d
100 1001	I	110 0101	e
100 1010	J	110 0110	f
100 1011	K	110 0111	g
100 1100	L	110 1000	h
100 1101	M	110 1001	i
100 1110	N	110 1010	j
100 1111	O	110 1011	k
101 0000	P	110 1100	1
101 0001	Q	110 1101	m

ASCII (Cont)

Bit Configuration	Symbol
110 1110	n
110 1111	o
111 0000	p
111 0001	q
111 0010	r
111 0011	s
111 0100	t
111 0101	u
111 0110	v
111 0111	w
111 1000	x
111 1001	y
111 1010	z
111 1011	{
111 1100	¬
111 1101	}
111 1110	|
111 1111	DEL

APPENDIX F

APPENDIX F

ANSWERS TO EXERCISES ON CHAPTER II

1. a. 0010 1101 1010 1110 is $2DAE_{16}$

   b. 1000 1111 1100 0111 is $8FC7_{16}$

   c. 1111 1111 1100 0000 is $FFC0_{16}$

   $2DAE_{16} = 11694_{10}$

   $8FC7_{16} = -28728_{10}$

   $FFC0_{16} = -63_{10}$

   $2DAE_{16} + 8FC7_{16} + FFC0_{16} =$

   $BD75_{16} + FFC0_{16} = BD36_{16}$

   No overflow was generated.

2. a. $4095_{10} = 0FFF_{16} = 0000\ 1111\ 1111\ 1111_2$

   b. $-17_{10} = FFEE_{16} = 1111\ 1111\ 1110\ 1110_2$

   c. $255_{10} = 00FF_{16} = 0000\ 0000\ 1111\ 1111_2$

3. By extending the sign. Consider the 8 bit positive number:

   00000111_2 - Decimal value 7

   ↑ Sign Bit. Extend the sign bit to the left 8 places - $0000\ 0000\ 0000\ 0111_2$ - same value of 7 but the number now occupies a 16-bit field.

   The same applies for a negative number. FE_{16} is -1 in an 8-bit field and $FFFE_{16}$ is also -1, but in a 16-bit field.

ANSWERS TO EXERCISES ON CHAPTER V - Shift and Skip Instructions

1. a. Shifts QA left 16 bits. This effectively switches the data in A to Q and Q to A.

   b. Will JUMP to P + 4 if the overflow indicator is on, and clear the indicator. Will go to P + 1 if overflow indicator is not on.

   c. Shifts the data in A right 18 bits. This will clear A if bit 15 was originally a 0 or will set A to all 1's if bit 15 was originally a 1. It also wastes time since the maximum necessary shift count to accomplish the same result is 15.

   d. Neither the A nor the Q register is specified so the instruction is essentially a time delay. It takes 1.1 + .1 x8 = 1.9 microseconds to execute and does nothing to the A or Q registers.

2. Q = 0000
   A = E3C4

ANSWERS TO EXERCISES ON CHAPTER V - Constant Mode of Addressing

1. a. A = 1059 Q = 0000

   b. A = 0024 Q = 0000

   c. A = 4142 Q = 4040

ANSWERS TO EXERCISES ON CHAPTER V - Absolute Mode of Addressing

1.
```
LDA-            $FF
ADD             =N$10
STA-            $FF
```

also

```
LDA-            I
ADD             =N$10
STA-            I
```

I can be used in place of $FF for storage reference class of instructions. One-word absolute mode is used since $FF is in the communications region.

2. a. 4016

   b. 003F

   c. 4016 (However, this is two-word absolute indirect which requires two core cells and 1.1 microseconds more time. Since the base address is in the communications region, one-word absolute indirect should be used.)

3. a. This will decode as C000; Δ will be 0, yielding not one-word absolute but the first word of constant mode. The base address for one-word absolute must be 01 to FF.

   b. One-word absolute could have been used since the base address is F3.

   c. Nothing. This is a legitimate example of two-word absolute indirect. Bit 15 of the second word (P+1) will be set because of the parentheses and bits 14-0 of P+1 will contain the address equivalent for TEST.

ANSWERS TO EXERCISES ON CHAPTER V - Relative Mode of Addressing

1. a. Forward, 31 hex locations from P.

   b. Backward, 1000_{16} or 4096_{10} from P+1, or $0FFF_{16}$ or 4095_{10} from P.

   c. Backward, one location back from P.

2. a. Two-word relative
   b. One-word absolute indirect
   c. Two-word absolute indirect
   d. One-word relative
   e. Constant. Two-word.
   f. Two-word relative indirect
   g. Constant. Two-word.
   h. One-word absolute

3. The communications region is fixed and will never move with the program. If the program moved and not the communications region, the relative distance which had been established would be worthless. Fixed areas of core are addressed absolutely, and those that move with the program are addressed relatively.

ANSWERS TO EXERCISES ON CHAPTER V - Indexing

1. a. B is used to indicate use of both Q and I index registers. Should be written LDA AB, B.

   b. Nothing, if A is defined as a legitimate symbol elsewhere in the program.

   c. No indexing with shift class instructions, only storage reference class.

2. a. A = 0000

   b. A = 0023

   c. A = 0023

3. It will loop forever since the reduced index register I value is never stored back. Need a STQ- I after the ADQ =N-1.

Answer to indexing problem:

a. $1234

b. $02ED

c. $2311

d. $1111

ANSWERS TO EXERCISES ON CHAPTER VI - Utility Assembler

1. a. Symbols for the EQU and BSS pseudo ops are defined by appearing in the address field.

   b. TAG is a doubly defined symbol. It cannot be an EXT.

   c. START should be an entry point. The program should look like:

```
Location          Opcode        Address         Comments

                  NAM           EXAMP
                  ENT           START
                  EQU           LED(720)
TAG               NUM           -72,$FFFF,72
                  BSS           TAG11(25)
                  EXT           LAD
START             LDA+          TAG
                  STA+          LAD
                  -
                  -
                  END           START
```

2. The common area cannot be preset.

3. Since the common area is fixed, all references to it can safely be made absolutely. This is necessary instead of a relative addressed mode for run anywhere programs.

4. When the reference is made to LIST+3, the operand is unpredictable. The loader skips over the LIST area at load time. A BZS should have been used. It would be worthwhile at this point to refer to reference 5 or 6 for more information concerning the loader since this information is not covered in this manual.

ANSWERS TO EXERCISES ON CHAPTER VI - Macro Assembler

1. a. FEE9
   002F

   b. 4142

   c. 0A00

   d. FFFE

2.
```
LDA        =N$1000
ADD        =N$1000
```

3.
```
TAG        ALF           Z,ERROR6 Z
TAG1       ALF           Z,LOGICAL UNIT8 Z
```

4. The symbol TE is declared local to the macro and cannot be called by the main program. Also, since the code is inserted in-line, the $7FF3 could be executed as an instruction.

ANSWERS TO EXERCISES ON CHAPTER VIII

1.

```
                    STA*        FIRST
                    ENA         $20
                    LDQ         =N$A1
                    OUT         -1
                    INQ         -1
                    INP         -1
                    SAN          1
                    JMP*        *-2
        LOOP        INP         -1
                    ALS          8
                    INP         -1
                    STA*        (FIRST)
                    RAO*        FIRST
                    JMP*        LOOP
        FIRST       ADC         0
```

a. This program is run anywhere.

b. Starting location for the checksum program is initially placed in the A register.

c. Reader will run out of tape and stop on a lost data condition.

APPENDIX G

APPENDIX G

SOLUTIONS TO PROBLEMS IN CHAPTER V

ADDRESSING PROBLEM

a. 1234

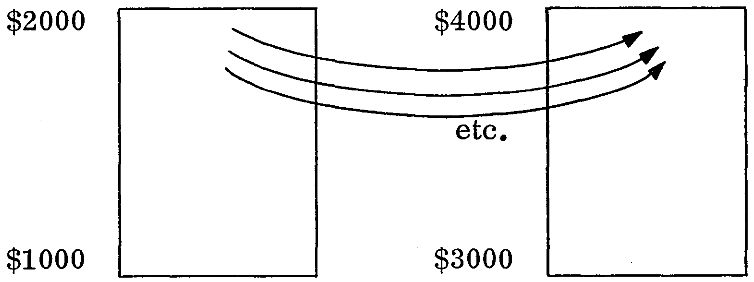
b. 02ED

c. 2311

d. 1111

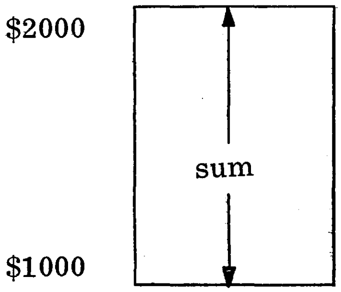
MOVE

The MOVE routine moves 1001_{16} numbers from \$1000-\$2000, inclusive, to \$3000-\$4000, inclusive.



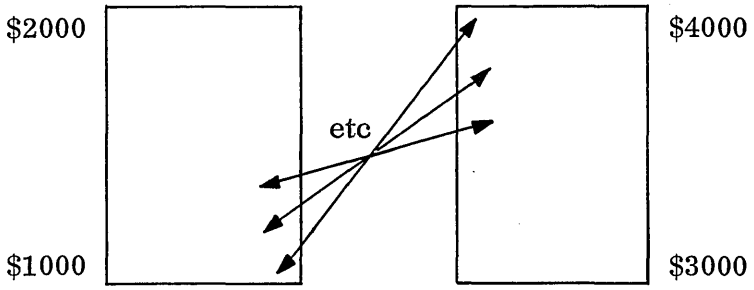
SUM

The SUM routine sums 1001_{16} numbers from \$1000-\$2000, inclusive. The answer is stored in \$3000.



CHNG

The CHNG routine swaps two arrays, inverting them. The contents of locations $1000-$2000 are swapped and inverted with the contents of locations $3000-$4000.



The swap is completed when the last address in ADDR has been decremented past $3000.

SUB

The SUB program picks up the actual data from the calling program and stores them in its locations SUBAG1, SUBAG2, and SUBAG3. The EOR =N$8000 put an indirect bit on the contents of location SUB (which in the example was $502). This caused the LDA* (SUB) to get the actual data, 10, etc. After picking up all the data, the subroutine updated the return address to $505 and ANDed off the extra indirect bit, AND =N$7FFF, so the return would be to the proper place.

The subroutine saved the registers when it was entered, and restored them when it exited, which most subroutines would do.

SORT

The SORT routine sorts 16 numbers, in ascending order, in core locations $500-$50F, inclusive. The method used is to push the smallest number to the top, the next smallest number to the next-to-top, etc.

CLRPB

The CLRPB subroutine clears protect bits on the core area from $2000-$4000, inclusive. The parameters passed by the calling routine are the last word address of the buffer (LWA) and the first word address (FWA). The subroutine will work for any addresses passed to it. It uses the A register and overflow indicator for loop control. When the last word address is subtracted from $7FFF in A, the number left will be such that when Q reaches the last word address +1 and the AAQ 0 instruction is executed, overflow will occur, and the exit will be made from the loop. The SOV 0 instruction before entering the loop turned off the overflow indicator in case it may have been on. The CLRPB subroutine must be a protected routine itself or the protect switch must be off.

CONVRT

The CONVRT subroutine converts a hexadecimal number which it receives in A to the ASCII codes for the decimal number. It packs the codes in a buffer, BUF, and returns to the caller with the buffer address in A.

The method used is to divide the number by 10, in hexadecimal arithmetic, and save the remainders. Each time a divide is done, the remainder in Q will be used to get the ASCII code for the number from a table, TAB.

It would have been possible in this example to simply add $30 to each remainder for the conversion. However, the table-look-up method is used for illustration. Also note that the routine puts the characters in one buffer, BUF1, then packs them in BUF for the caller.

The CONTST program checks out CONVRT by calling it to convert a number $37C0, then writing the answer on the teletypewriter in ASCII.

SOLUTIONS TO PROBLEMS IN CHAPTER VI

```
VALUE       PROBLEM

            NAM         TEST
            COM         DUMMY(10),X(10)
            DAT         DUM(6),COUNT(1)
            EQU         LPMASK($2)
            ENT         START
            EXT*        VALUE
MASK        BZS         MASK(1)
START       0           0
            CLR         Q
            STQ         COUNT
            LDA         VALUE
            AND-        LPMASK+6
            ALS         8
            STA*        MASK
            ENQ         9
SEARCH      LDA+        X,Q
            AND         =N$3F00
            EOR*        MASK
            SAN         2
            RAO         COUNT
            SQZ         EXIT-*-1
            INQ         -1
            JMP*        SEARCH
EXIT        JMP*        (START)
            END
```

NAM gives the program a name. ENT gives it an entry point. START is where execution begins, but it does not appear on the END card, implying that it is a subroutine. COM declares the X's in COMMON; COUNT declares the 7th word of the data block for the answer. DUMMY and DUM space over locations not used in this program. The EXT* declares VALUE external (it must be an entry point in the other program), and the * implies it is relative. The BZS initializes MASK to 0. The EQU declares that LPMASK is location $2.

INI MACRO

The two listings on the following pages show one way to write the INI macro. The first listing does not have the macro expanded, while the second does (M option).

The test routine was run with the STOP switch set. When the computer stopped, the Q register was selected on the console for display. It contained a 6, to indicate that the macro worked.

The INI is not exactly like an INA or INQ because of the size of operands allowed. Line 0008 has an ADD =N'N' instead of an INA 'N'.

```
0001                        NAM     INIMAC
0002                 INI    MAC     N
0003                        LOC     SA
0004                        JMP*    *+2
0005                 'SA'   NUM     0
0006                        STA*    'SA'
0007                        LDA-    I
0008                       ADD      =N'N'
0009                        STA-    I
0010                        LDA*    'SA'
0011                        EMC
0012                        ENT     INIMAC
0013 P0000 0000      INIMAC 0       0
0014 P0001 0842             CLR     Q
0015 P0002 40FF             STQ-    I
0016                        INI     6
0016 P0003 1802
0016 P0004 0000
0016 P0005 68FE
0016 P0006 C0FF
0016 P0007 8000
     P0008 0006
0016 P0009 60FF
0016 P000A C8F9
0017 P000B E0FF             LDQ-    I
0018 P000C 0000             SLS
0019                        END     INIMAC
```

```
I      00FF   INIMAC   0000P [00      0004P
```

```
0001                                NAM      INIMAC
0002                     INI        MAC      N
0003                                LOC      SA
0004                                JMP*     *+2
0005                     'SA'       NUM      0
0006                                STA*     'SA'
0007                                LDA-     I
0008                               ADD       =N'N'
0009                                STA-     I
0010                                LDA*     'SA'
0011                                EMC
0012                                ENT      INIMAC
0013 P0000 0000          INIMAC     0        0
0014 P0001 0842                     CLR      Q
0015 P0002 40FF                     STQ-     I
0016                                INI      6
0016 P0003 1802                     JMP*     *+2
0016 P0004 0000          [00       NUM      0
0016 P0005 68FE                     STA*     [00
0016 P0006 C0FF                     LDA-     I
0016 P0007 8000                    ADD       =N6
     P0008 0006
0016 P0009 60FF                     STA-     I
0016 P000A C8F9                     LDA*     [00
0017 P000B E0FF                     LDQ-     I
0018 P000C 0000                     SLS
0019                                END      INIMAC
```

```
I        00FF   INIMAC   0000P [00        0004P
```

SOLUTION TO PROBLEMS IN CHAPTER VIII

CKASSM PROBLEM (8.4)

The first INDIR request, line 13, is for the write request at line 34, out of the buffer MSG, line 25, on the teletypewriter. "NEXT MESSAGE SHOULD INDICATE VERIFICATION."

The next, at line 15, is to start the write request at line 32 to write a message from MSG1, line 26, on the disk. "MACRO ASSEMBLER ON 1700 NOW OK." This message will be written on the disk concurrently with the first message which is going out on the TTY.

At line 16, a status request is made to wait for the disk write to finish. Then, at line 19, the message on the disk is read back in, line 28, into a different buffer, BUF, which is a BZS at line 27. At line 20, status is taken from line 28 to wait for completion of the disk read. Finally, this message is transferred out of BUF to the teletypewriter at line 23, after which an exit is made at line 24.

The program works out the assembler by the assortment of requests used to transfer the message around. Note that no completion routines are used; it will be obvious that if the two messages come out, it worked; and if they don't, it didn't.

The errors which could be found are:

1. SQN 1 at lines 17 and 21. This is the main error and the instruction effectively doesn't do anything. It was intended to loop on the indirect status request until the operation in progress bit (bit 15 of Q) became clear. To accomplish that, a SQP instruction should have been used. Since Q is word 8 of the disk physical device table, it will never be a whole word of zero (upon which the SQN could be used). Only bit 15 should be checked for zero. Therefore, as the program is set up, control falls right through to initiate the disk read at line 19, then the teletypewriter write at line 23.

   The program works because of the speed of the peripherals involved and the threading of requests onto the driver for each logical unit. The correct message is written, not garbage.

   What actually happens (if the SQN's are used instead of SQP's) is:

   a. The first TTY message is initiated.

   b. The disk write is initiated.

   c. The disk read is initiated; it is threaded onto the disk driver <u>after</u> the disk write, since both are at priority 0. Therefore, the read will not be done until the write is finished.

   d. The second TTY write is initiated and is threaded on behind the first write, again since the priority is the same (RP = 0). That is why the second line comes out after the first, as it should.

   e. The second line does not contain garbage because the TTY is slow compared with the disk; the disk buffer has been written and read back in before the TTY driver gets ready to write it out. Any change in hardware or priorities involved could cause a mess.

2. The next item to note is the binary write on the teletypewriter at line 23. Normally an ASCII write would be used. No damage is done because the words being written already contain ASCII characters which can be sent directly to the teletypewriter.

3. Note the formatted write and read on the disk. This is not an error but implies that the disk sector driver is in the system, not the disk word driver.

4. Note the disk sector address words which must be inserted after the macros at lines 30 and 33. They indicate sector 1 in the scratch area. This is not an error.

The version of the CKASSM routine which is used to check out the macro assembler contains SQN instructions at P0007 and P000D. This is probably to test the status request itself and then also to check out the operation of MSOS.

SOLUTIONS TO PROBLEMS IN CHAPTER XI

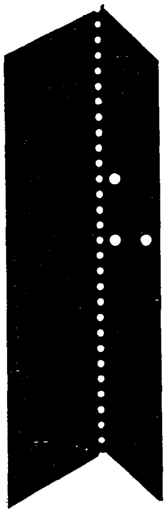
SOLUTION TO RUN ANYWHERE PROBLEM

Program AVERAGE is written as a nondestructive run anywhere subroutine which may be called to compute an average. The buffer address in the calling routine is X, and the number of words is 10. Note that all of the addressing in the program is relative.

```
                            NAM    AVERAGE
                            EXT*   X             RELATIVE EXTERNAL X
                            ENT    AVG
P0000  0001                 BZS    OVFL(1)
P0001  0000      AVG        0      0
P0002  0C09                 ENQ    9             N WORDS 9
P0003  0A00                 ENA    0             CLEAR A AND OVFL
P0004  68FB                 STA*   OVFL          RELATIVE TO OVFL
P0005  01A0                 SOV    0
P0006  8A00 X    LOOP       ADD    X,Q           2  WORD RELATIVE EXT
P0007  7FFFX
P0008  01B3                 SNO    TEST-*-1
P0009  D8F6                 RAO*   OVFL          RELATIVE TO OVFL
P000A  A000                 AND    =N$7FFF
P000B  7FFF
P000C  0142      TEST       SQZ    AV-*-1
P000D  0DFE                 INQ    -1
P000E  18F7                 JMP*   LOOP          RELATIVE TO LOOP
P000F  E8F0      AV         LDQ*   OVFL          RELATIVE TO OVFL
P0010  0FC1                 ALS    1
P0011  0F61                 LRS    1
P0012  3000                 DVI    =N10
P0013  000A
P0014  1CEC                 JMP*   (AVG)         RELATIVE TO AVG
                            END
```

PROB1 calls AVERAGE and punches out the answer, 4, and remainder, 5. The program is run anywhere, and note the relative addressing used in the punch request. ANS is before the request and COMPL is after the request.

```
0001                             NAM    PROB1
0002                             EXT*   AVG,IOERR              RELATIVE EXTERNAL AVG
0003                             ENT    START,X
0004  P0000 0001      X          NUM    1,2,3,4,5,6,7,8,9,0
      P0001 0002
      P0002 0003
      P0003 0004
      P0004 0005
      P0005 0006
      P0006 0007
      P0007 0008
      P0008 0009
      P0009 0000
0005  P000A 0003      ANS        BZS    ANS(3)
0006        00EA                 EQU    ADISP($EA)
0007  P000D 5800 X    START      RTJ    AVG                    TWO-WORD RELATIVE EXT
      P000E 7FFF X
0008  P000F 6800                 STA    ANS                    RELATIVE TO ANS
      P0010 FFF9
0009  P0011 4800                 STQ    ANS+2
      P0012 FFF9
0010  P0013 54F4                 RTJ-   ($F4)                  PUNCH OUT ANS AND REM
0011  P0014 0501                 NUM    $0501                  SET  X  BIT
0012  P0015 0007                 ADC*   COMPL+1                RELATIVE TO COMPL (FWD)
0013  P0016 0000                 NUM    0
0014  P0017 0003                 NUM    $0003
0015  P0018 0003                 NUM    3
0016  P0019 7FF5                 ADC    ANS-*+5                RELATIVE TO ANS
0017  P001A 14EA                 JMP-   (ADISP)
0018  P001B 0162      COMPL      SQP    2
0019  P001C 5800 X               RTJ    IOERR
      P001D 7FFF X
0020  P001E 18FF                 NUM    $18FF                  HANG WHEN FINISHED
0021                             END    START
```



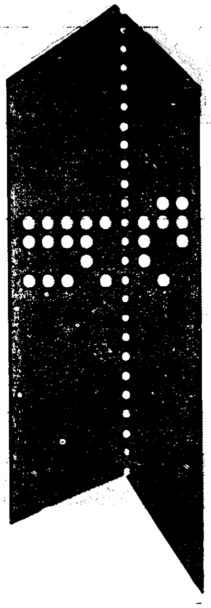
Here is an error example of incorrect relative addressing used in the punch request. Note at line 16 that the ADC* ANS+5 assembles as FFF5, because ANS comes before the ADC in the program. A positive increment is required, relative to the first word of the parameter string, so it should have been 7FF5. The ADC ANS-*+5 should have been used.

In debugging this program, the punch tape was studied. (Be sure to get it right side up.) Note that the codes punched are:

```
03
FF
F5
14
EA
```

By studying the output, it was noticed that these are the codes for part of the program, indicating that the buffer ANS was not output, but program codes were. Hence the discovery of the FFF5 at P0019.

```
0001                            NAM   PROB1
0002                            EXT*  AVG,IOERR
0003                            ENT   START,X
0004 P0000 0001    X            NUM   1,2,3,4,5,6,7,8,9,0
     P0001 0002
     P0002 0003
     P0003 0004
     P0004 0005
     P0005 0006
     P0006 0007
     P0007 0008
     P0008 0009
     P0009 0000
0005 P000A 0003    ANS          BZS   ANS(3)
0006       00EA                 EQU   ADISP($EA)
0007 P000D 5800  X START        RTJ   AVG
     P000F 7FFF  X
0008 P000F 6800                 STA   ANS
     P0010 FFF9
0009 P0011 4800                 STQ   ANS+2
     P0012 FFF9
0010 P0013 54F4                 RTJ-  ($F4)
0011 P0014 0501                 NUM   $0501
0012 P0015 0007                 ADC*  COMPL+1
0013 P0016 0000                 NUM   0
0014 P0017 0003                 NUM   $0003
0015 P0018 0003                 NUM   3
0016 P0019 FFF5                 ADC*  ANS+5
0017 P001A 14EA                 JMP-  (ADISP)
0018 P001B 0162    COMPL        SQP   2
0019 P001C 5800  X              RTJ   IOERR
     P001D 7FFF  X
0020 P001E 18FF                 NUM   $18FF
0021                            END   START
```



In order to verify that the programs were indeed run anywhere and nondestructive and did link correctly, coding was inserted to move them (after they ran one time) to higher core and run again. Two identical answers would indicate that the programs did run correctly.

The following example includes coding added which will move 100_{10} words, beginning at START, to $4000, and then jump to $4000 to reexecute. This continues repeatedly. Since the program lengths are 28_{16} and $1D_{16}$, for a total of 45_{16}, the 100_{10} (64_{16}) words moved is sufficient.

Another method would be to use the contents of locations $F7 and $ED to find the core address and program length, and use them in the move.

When checkout methods such as this are used, remember that if the breakpoint package is in core it is physically located immediately above the last subroutine. Therefore, it would be wise not to move the programs on top of the breakpoint package and wipe it out.

```
0001                               NAM   PROB1
0002                               EXT*  AVG,IOERR
0003                               ENT   START,X
0004         00EA                  EQU   ADISP($EA)
0005 P0000 5800 X START            RTJ   AVG
     P0001 7FFF X
0006 P0002 6800                    STA   ANS
     P0003 0024
0007 P0004 4800                    STQ   ANS+2
     P0005 0024
0008 P0006 54F4                    RTJ-  ($F4)
0009 P0007 0501                    NUM   $0501
0010 P0008 0007                    ADC*  COMPL+1
0011 P0009 0000                    NUM   0
0012 P000A 0003                    NUM   $0003
0013 P000B 0003                    NUM   3
0014 P000C 0020                    ADC   ANS-*+5
0015 P000D 14EA                    JMP-  (ADISP)
0016 P000E 0162   COMPL            SQP   2
0017 P000F 5800 X                  RTJ   IOERR
     P0010 7FFF X
0018 P0011 0A00                    ENA   0           }
0019 P0012 60FF                    STA-  I           }
0020 P0013 C9EC   LOOP             LDA*  START,I     }
0021 P0014 6500                    STA+  $4000,I     }
     P0015 4000                                      }    MOVE
0022 P0016 D0FF                    RAO-  I           }
0023 P0017 C0FF                    LDA-  I           }
0024 P0018 099C                    INA   -99         }
0025 P0019 0101                    SAZ   OUT         }
0026 P001A 18F8                    JMP*  LOOP        }
0027 P001B 1400   OUT              JMP+  $4000
     P001C 4000
0028 P001D 0001   X                NUM   1,2,3,4,5,6,7,8,9,0
     P001E 0002
     P001F 0003
     P0020 0004
     P0021 0005
     P0022 0006
     P0023 0007
     P0024 0008
     P0025 0009
     P0026 0000
0029 P0027 0003   ANS              BZS   ANS(3)
0030                               END   START
```

To illustrate the use of the conversion routine, the following program MAIN calls AVG to compute an average, then calls CONVRT to convert the answer (which was returned in A by AVG). CONVRT returns a 3-word buffer address of the ASCII characters (in A) which the main program then stores down in the write request.

Note that this is an absolute buffer address. The completion address, COMPL, must also be absolute, yet it is in the run anywhere program MAIN. For that reason, the program must absolutize the completion address at each run and store it at C in the write request.

After the answer is computed once, MAIN calls the MOVE subroutine to move everything except CONVRT up to higher core locations, immediately following the original programs. Absolute addressing is used in MAIN to CONVRT so that the original copy of CONVRT is used for conversion, since it is not a run anywhere program.

This would simulate a mass memory module using a core resident subroutine which always remained in the same place. Even if CONVRT had been moved, the original copy would still have been the one called.

Note also that the flag used to control the move is addressed absolutely. This is so the original flag in core will be used: set by one routine and checked by another. This will simulate run anywhere routines communicating with each other through a core resident flag cell. In the current example, the flag could have been addressed relatively.

```
0001                         NAM         MAIN
0002                   *                                         FRAN GIACOBBE - PHILA
0003                   *                                         CLASS PROBLEM NO. 1 P,
0004                         ENT         MAIN,X,MVFLAG
0005                         EXT*        AVG,MOVE
0006                         EXT         CONVRT
0007                   *
0008                   *  MAIN MUST BE LOADED FIRST, CONVRT LAST, FOR CHECKOUT
0009                   *  MAIN ENTRY POINT MUST BE FIRST LOCATION IN MAIN
0010                   *
0011 P0000 5801        MAIN   RTJ*        *+1                ABSOLUTIZE COMPL ADDR
0012 P0001 0000        T      0           0
0013 P0002 C8FE               LDA*        *-1
0014 P0003 8000               ADD         =XCOMPL-T
     P0004 0012
0015 P0005 6808               STA*        C
0016 P0006 5800 X             RTJ         AVG                GO GET AVERAGE
     P0007 7FFF X
0017 P0008 5400 X             RTJ+        CONVRT             GO CONVERT ANSWER
     P0009 7FFF X
0018 P000A 6807               STA*        WRITE+6            BUFFER ADDRESS
0019 P000B 54F4        WRITE  RTJ-        ($F4)
0020 P000C 0C01               NUM         $0C01              GO WRITE ANSWER
0021 P000D 0000        C      ADC         0
0022 P000E 0000               NUM         0
0023 P000F 1004               NUM         $1004
0024 P0010 0003               NUM         3
0025 P0011 0000               ADC         0
0026 P0012 14EA               JMP-        ($EA)
0027 P0013 0161        COMPL  SQP         CKMOVE
0028 P0014 18FF               NUM         $18FF              HANG IF ERROR
0029 P0015 C400        CKMOVE  LDA+       MVFLAG             FIRST OR SECOND RUN?
     P0016 0025 P
0030 P0017 0101               SAZ         GOMOVE-*-1
0031 P0018 14EA               JMP-        ($EA)
0032 P0019 1800 X      GOMOVE JMP         MOVE
     P001A 7FFF X
0033 P001B 15F9        X      NUM         $15F9,$7FFF,$65,2,0,$100D,8,$12,4,3
     P001C 7FFF
     P001D 0065
     P001E 0002
     P001F 0000
     P0020 100D
     P0021 0008
     P0022 0012
     P0023 0004
     P0024 0003
0034 P0025 0001        MVFLAG BZS         MVFLAG(1)
0035                          END         MAIN



I        00FF  MAIN     0000P  X        001BP  MVFLAG  0025P  T        0001P
WRITE    000BP C        000DP  COMPL    0013P  CKMOVE  0015P  GOMOVE   0019P
CONVRT   0009X MOVE     001AX  AVG      0007X
```

```
0001                                NAM       AVERAGE
0002                                EXT*      X                   BUFFER IN MAIN PGM
0003                                ENT       AVG
0004 P0000 0001     OVFL            BZS       OVFL(1)
0005 P0001 0000     AVG             0         0
0006 P0002 0C09                     ENQ       9                   N WORDS 10
0007 P0003 0A00                     ENA       0                   CLEAR A AND OVFL
0008 P0004 68FB                     STA*      OVFL
0009 P0005 01A0                     SOV       0                   TURN OFF OVERFLOW INDICATO
0010 P0006 8A00 X LOOP              ADD       X,Q                 ADD UP DATA
     P0007 7FFF X
0011 P0008 01B3                     SNO       TEST-*-1
0012 P0009 D8F6                     RAO*      OVFL                IF OVERFLOW
0013 P000A A000                     AND       =N$7FFF
     P000B 7FFF
0014 P000C 0142     TEST            SQZ       AV-*-1
0015 P000D 0DFE                     INQ       -1
0016 P000E 18F7                     JMP*      LOOP
0017 P000F E8F0     AV              LDQ*      OVFL                COMPUTE AVERAGE
0018 P0010 0FC1                     ALS       1
0019 P0011 0F61                     LRS       1
0020 P0012 3000                     DVI       =N10
     P0013 000A
0021 P0014 1CEC                     JMP*      (AVG)
0022                *  LEAVE ANSWER IN A AND REMAINDER IN Q AND RETURN TO CALLER
0023                                END


I       00FF  AVG       0001P OVFL      0000P LOOP      0006P TEST      000CP
AV      000FP X         0007X
```

```
0001                            NAM     MOVE
0002                            ENT     MOVE
0003                            EXT     MVFLAG
0004        0022                EQU     ZERO($22)
0005 P0000 0000    MOVE         0       0
0006 P0001 C0ED                 LDA-    $ED           FIGURE OUT N WORDS
0007 P0002 90F7                 SUB-    $F7
0008 P0003 9000                 SUB     =N$44         SUBTRACT OFF SIZE OF CONVR
     P0004 0044
0009               *                                  SO IT WONT GET MOVED
0010 P0005 0822                 TRA     Q
0011 P0006 C6F7    MOVLP        LDA-    ($F7),Q       MOVE IT
0012 P0007 66ED                 STA-    ($ED),Q
0013 P0008 0DFE                 INQ     -1
0014 P0009 0141                 SQZ     OUT-*-1
0015 P000A 18FB                 JMP*    MOVLP
0016 P000B D400 X OUT           RAO+    MVFLAG        SET MOVED FLAG
     P000C 7FFF X
0017 P000D E0ED                 LDQ-    $ED           FIGURE OUT JUMP ADDRESS
0018 P000E 0D01                 INQ     1
0019 P000F 1622                 JMP-    (ZERO),Q      JUMP TO MAIN
0020                            END
```

```
I         00FF  MOVE      0000P ZERO      0022  MOVLP     0006P OUT       000BP
MVFLAG    000CX
```

```
0001                       NAM       CONVRT
0002 P0000 0001            BSS       SAVEQ(1),SAVEI(1),SAVEA(1)
     P0001 0001
     P0002 0001
0003 P0003 0003   BUF      BSS       BUF(3)
0004 P0006 0006   BUF1     BSS       BUF1(6)
0005 P000C 002B   SIGN     NUM       $2B,$2D
     P000D 002D
0006 P000E 0030   TAB      NUM       $30,$31,$32,$33,$34,$35,$36,$37,$38,$39
     P000F 0031
     P0010 0032
     P0011 0033
     P0012 0034
     P0013 0035
     P0014 0036
     P0015 0037
     P0016 0038
     P0017 0039
0007                       ENT       CONVRT
0008 P0018 0000   CONVRT   0         0
0009 P0019 48E6            STQ*      SAVEQ
0010 P001A E0FF            LDQ-      I
0011 P001B 48E5            STQ*      SAVEI
0012 P001C 0842            CLR       Q
0013 P001D 40FF            STQ-      I
0014 P001E 0122            SAP       POS
0015 P001F 0D01            INQ       1
0016 P0020 0864            TCA       A
0017 P0021 EAEA   POS      LDQ*      SIGN,Q
0018 P0022 48E8            STQ*      BUF1+5
0019 P0023 E8EA            LDQ*      TAB+0
0020 P0024 48E1            STQ*      BUF1
0021 P0025 E000            LDQ       =N$20
     P0026 0020
0022 P0027 48DF            STQ*      BUF1+1
0023 P0028 48DF            STQ*      BUF1+2
0024 P0029 48DF            STQ*      BUF1+3
0025 P002A 48DF            STQ*      BUF1+4
0026 P002B 0842   LOOP     CLR       Q
0027 P002C 0106            SAZ       OUT
0028 P002D 3000            DVI       =N10
     P002E 000A
0029 P002F EADE            LDQ*      TAB,Q
0030 P0030 49D5            STQ*      BUF1,I
0031 P0031 D0FF            RAO-      I
0032 P0032 18F8            JMP*      LOOP
0033 P0033 40FF   OUT      STQ-      I
0034 P0034 0C05            ENQ       5
0035 P0035 CAD0   BACK     LDA*      BUF1,Q
0036 P0036 0FC8            ALS       8
0037 P0037 0DFE            INQ       -1
0038 P0038 8ACD            ADD*      BUF1,Q
0039 P0039 69C9            STA*      BUF,I
0040 P003A D0FF            RAO-      I
0041 P003B 0DFE            INQ       -1
```

```
0042 P003C 0171               SQM         DONE
0043 P003D 18F7               JMP*        BACK
0044 P003E E8C1    DONE       LDQ*        SAVEQ
0045 P003F C8C1               LDA*        SAVEI
0046 P0040 60FF               STA-        I
0047 P0041 C000               LDA         =XBUF
     P0042 0003 P
0048 P0043 1CD4               JMP*        (CONVRT)
0049                          END
```

```
I        00FF   SAVEQ    0000P SAVEI    0001P SAVEA    0002P BUF      0003P
BUF1     0006P  SIGN     000CP TAB      000EP CONVRT   0018P POS      0021P
LOOP     002BP  OUT      0033P BACK     0035P DONE     003EP
```

ADDRESSES WHERE
PROGRAMS LOADED

```
MAIN     236C
AVERAG   2392
MOVE     23A7
CONVRT   23B7
```

MAP

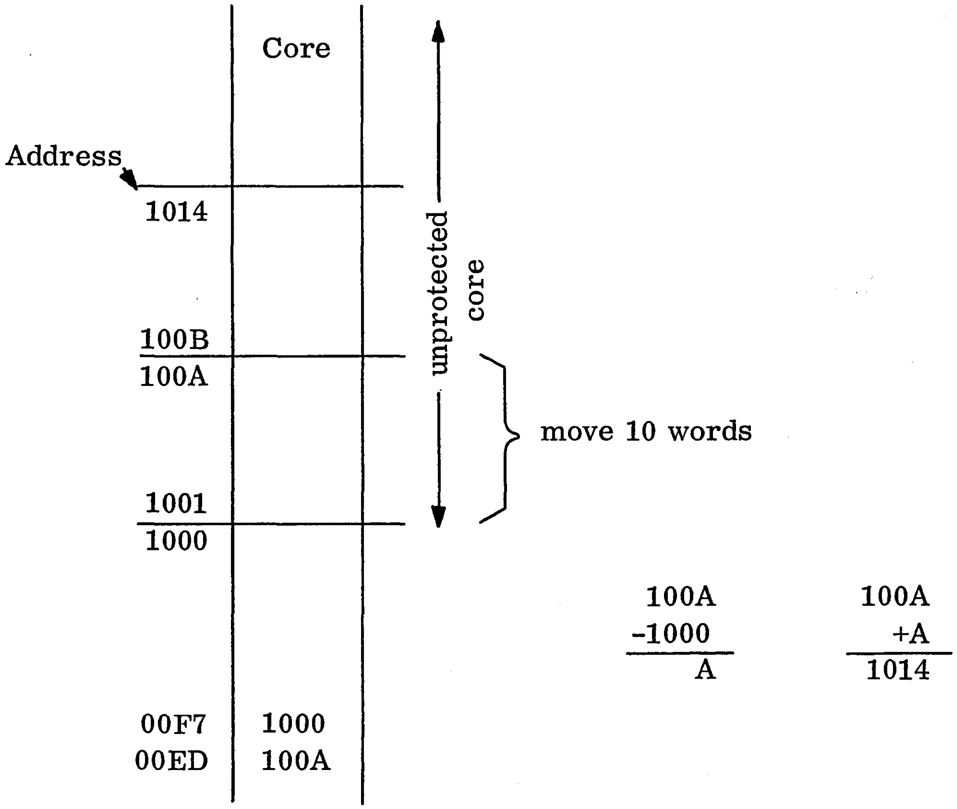
```
1 ENTRY POINT TABLE-
  MAIN      236C     X          2387    MVFLAG   2391     CONVRT   23CF
  MOVE      23A7     AVG        2393
```

TTY PRINTOUT

```
*
MI
*P
J
*K, I13, P6
J
*ASSEM
OPTIONS  LX
J
*P
J
*L, 8
J
*X
+  4263  ← Answers
+  4263
J



MI
*P
J
*UTOPIA



E  *
CKDISK - ENTER CKD/  FOR HEADING

ADH, 15F9, 7FFF/
95F8
ADH, 95F8, 65/
965D
ADH, 965D, 2/
965F
ADH, 965F, 100D/
A66C
ADH, A66C, 8/
A674
ADH, A674, 12/
A686
ADH, 4, A686/
A68A
ADH, A68A, 3/
A68D
```

Utopia was used to sum the hex numbers, to check out the answer. The sum is A68D$_{16}$ = 42637$_{10}$. Divide by 10$_{10}$ for a decimal answer of 4263. The remainder 7 was not considered when the answer was printed. Note that the numbers used did generate overflow in the sum, thereby checking out the average routine. (The average routine would work only for positive numbers.)

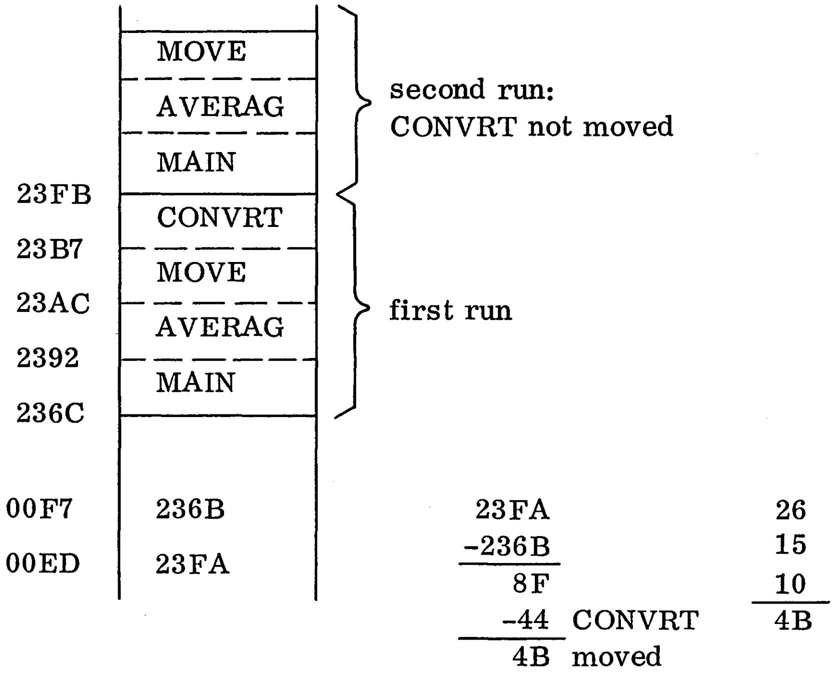
In analyzing and planning the move portion, the following simple method could be used with imaginary core addresses:



$$\begin{array}{r} \text{100A} \\ -\text{1000} \\ \hline \text{A} \end{array} \qquad \begin{array}{r} \text{100A} \\ +\text{A} \\ \hline \text{1014} \end{array}$$

By drawing a simple picture to move only 10 words, beginning at 1001 and showing the contents of $ED and $F7 (which will be used for indirect addressing) it canbe determined that the program should subtract the contents of $F7 from $ED and use that answer, $A, for the index in Q. The store through $ED would be analyzed the same way: ($ED) + $A = $1014. By looking at the picture, one can see that Q should go from $A to 1; hence, the skip out of the loop should be after the last move when Q equaled 1.

The actual addresses involved after loading were:



```
 23FA                 26
-236B                 15
   8F                 10
  -44  CONVRT         4B
   4B  moved
```

The total number of locations involved was $8F_{16}$. Less CONVRT, left $4B_{16}$ to move. Q was indexed from $4B_{16}$ (the last location in MOVE) through 1 (the first location in MAIN).

Solution to the AVG Reentrant Problem (11.3.2)

```
0001                          NAM  AVG
0002                          ENT  AVG
0003        00BB              EQU  AVOLA($BB),AVOLR($BA),ZERO($22),LPMASK($2)
            00BA
            0022
            0002
0004  P0000 0000   AVG        0    0
0005  P0001 0500              IIN
0006  P0002 54BB              RTJ- (AVOLA)
0007  P0003 0006              NUM  6
0008  P0004 0400              EIN
0009  P0005 C8FA              LDA* AVG             SAVE RETURN ADDRESS
0010  P0006 6103              STA- 3,I
0011  P0007 0844              CLR  A
0012  P0008 6104              STA- 4,I             ZERO OVERFLOW CELL
0013  P0009 F101              ADQ- 1,I             LWA+1 IN Q
0014  P000A 4105              STQ- 5,I             SAVE IN VOLA+5
0015  P000B 01A0              SOV  0               TURN OFF OVERFLOW
0016  P000C E4FF   LOOP       LDQ- (I)             FWA IN Q
0017  P000D 8622              ADD- (ZERO),Q        ADD DATA TO A
0018  P000E 01B2              SNO  TEST-*-1
0019  P000F D104              RAO- 4,I             COUNT OVERFLOW
0020  P0010 A011              AND- LPMASK+15       AND OUT SIGN BIT
0021  P0011 0D01   TEST       INQ  1               UPDATE ADDRESS
0022  P0012 4522              STQ- (ZERO),I        SAVE NEXT ADDRESS
0023  P0013 0852              TCQ  Q               COMPLEMENT NEXT ADDRESS
0024  P0014 F105              ADQ- 5,I             ADD LWA+1
0025  P0015 0141              SQZ  DV-*-1          FINISHED WHEN MATCH
0026  P0016 18F5              JMP* LOOP
0027  P0017 E104   DV         LDQ- 4,I             PICK UP OVERFLOW
0028  P0018 0FC1              ALS  1               SQUEEZE OUT SIGN BIT
0029  P0019 0F61              LRS  1
0030  P001A 3101   DIV        DVI- 1,I             DIVIDE FOR AVERAGE
0031  P001B 6101              STA- 1,I             RETURN ANSWER IN A
0032  P001C 4500              STQ+ 0,I             RETURN REMAINDER IN Q
      P001D 0000
0033  P001E C103   EXIT       LDA- 3,I             RESCUE RETURN ADDRESS
0034  P001F 0500              IIN
0035  P0020 68DF              STA* AVG
0036  P0021 54BA              RTJ- (AVOLR)
0037  P0022 0400              EIN
0038  P0023 1CDC              JMP* (AVG)
0039                          END
```

5	LWA+1
4	OVERFLOW
3	RETURN
2	I
1	A (N WORDS)
(I)+0	Q (FWA)

The following routine, NAM AVGTST, was used to check out the subroutine to see if it returned the correct answer. It only checks the answer and does not check the reentrancy. It asked AVG for an average of 9 words of a 10-word buffer X, and then punched the average 4 and remainder 0.

The method used to run the routine in the background, since VOLA is a protected routine, was:

1. Assemble and load AVGTST and AVG under MSOS. Do not execute yet.
2. Turn off protect switch on console.
3. Turn off disk (to protect the system).
4. Set P on the console to the beginning address of AVGTST and RUN.

This method could be used to check out any routine in the background which links to system routines in protected core. If the system in core gets clobbered, the image on the disk is intact.

```
0001                               NAM      AVGTST
0002                               ENT      X, BEGIN
0003                               EXT*     AVG
0004   P0000   0A09      BEGIN     ENA      9                             NUMBER OF WORDS
0005   P0001   E000                LDQ      =XX                           BUFFER ADDRESS
       P0002   0011 P
0006   P0003   5800 X              RTJ      AVG
       P0004   7FFFX
0007   P0005   6816                STA*     TEMP                          SAVE ANSWER
0008   P0006   4816                STQ*     TEMP+1                        SAVE REMAINDER
0009                     XA        WRITE    3, XB-XA-1, TEMP-XA-1, 2, B, 0, 1, A, X
0009   P0007   54F4
0009   P0008   0501
0009   P0009   0008
       P000A   0000
0009   P000B   0003
0009   P000C   0002
       P000D   0013
0010                               EXIT
0010   P000E   54F4
0010   P000F   0A00
0011   P0010   14EA      XB        JMP-     ($EA)
0012   P0011   0000      X         NUM      0, 1, 2, 3, 4, 5, 6, 7, 8, 9
       P0012   0001
       P0013   0002
       P0014   0003
       P0015   0004
       P0016   0005
       P0017   0006
       P0018   0007
       P0019   0008
       P001A   0009
0013   P001B   0002      TEMP      BSS      TEMP(2)
0014                               END      BEGIN



                                            ANSWER: 4
```

After testing out AVG to see if it gives an answer, it is then necessary to check out its reentrancy. In the following computer run, two programs (PGMA and PGMB) were set up, each to call AVG as a subroutine. Links and a flag were added to the routines for test purposes only. PGMA calls AVG and AVG begins its calculation. It then checks a flag and causes a pseudo interrupt of itself, and causes PGMB to begin its run. PGMB calls AVG, gets an answer, punches it, and returns control back to the location in AVG where the pseudo interrupt occurred. AVG completes its calculation for PGMA, returns to PGMA, and PGMA punches its answer and hangs.

Again, the programs are loaded under MSOS. Then the protect switch and disk are turned off while they execute.

This method could be used as a skeleton to check a routine's reentrancy.

The coding for testing reentrancy is marked by brackets in the example.

```
0001                                NAM      PGMA
0002                                ENT      PGMA
0003                                EXT      AVG
0004    P0000   0000      PGMA      0        0
0005    P0001   0A0A                ENA      10                 10 NUMBERS
0006    P0002   E000                LDQ      =XX                BUFFER ADDRESS X
        P0003   000F P
0007    P0004   5400 X              RTJ+     AVG                COMPUTE AVERAGE
        P0005   7FFFX
0008    P0006   6813                STA*     ANS
0009    P0007   54F4                RTJ-     ($F4)              PUNCH ANSWER
0010    P0008   0401                NUM      $0401
0011    P0009   0000                ADC      0
0012    P000A   0000                NUM      0
0013    P000B   0003                NUM      $0003
0014    P000C   0001                NUM      1
0015    P000D   0019 P              ADC      ANS
0016    P000E   18FF                NUM      $18FF              HANG
0017    P000F   0000      X         NUM      0,1,2,3,4,5,6,7,8,9
        P0010   0001
        P0011   0002
        P0012   0003
        P0013   0004
        P0014   0005
        P0015   0006
        P0016   0007
        P0017   0008
        P0018   0009
0018    P0019   0001      ANS       BSS      ANS(1)
0019                                END      PGMA
```

ANSWER: 4

```
0001                                 NAM     AVG
0002                                 ENT     AVG
0003                                 ENT     FINI
0004                                 EXT     PGMB
0005             00BB                EQU     AVOLA($BB), AVOLR($BA), ZERO($22), LPMASK($2)
                 00BA
                 0022
                 0002
0006    P0000    0000      AVG       0       0
0007    P0001    0500                IIN
0008    P0002    54BB                RTJ-    (AVOLA)
0009    P0003    0007                NUM     7                ONE MORE WORD VOLATILE
0010    P0004    0400                EIN                                      NEEDED
0011    P0005    C8FA                LDA*    AVG
0012    P0006    6103                STA-    3,I
0013    P0007    0844                CLR     A
0014    P0008    6014                STA-    4,I
0015    P0009    F101                ADQ-    1,I
0016    P000A    4105                STQ-    5,I
0017    P000B    01A0                SOV     0
0018    P000C    E4FF      LOOP      LDQ-    (I)
0019                       *
0020                       *
0021    P000D    6106                STA-    6,I
0022    P000E    C820                LDA*    FLAG
0023    P000F    0115                SAN     C
0024    P0010    0A01                ENA     1
0025    P0011    681D                STA*    FLAG             FIRST OR SECOND CALL?
0026    P0012    C106                LDA-    6,I
0027    P0013    1400 X              JMP+    PGMB
        P0014    7FFF X
0028                       *
0029    P0015    C106      C         LDA-    6,I
0030    P0016    0000      FINI      0       0                RETURN AFTER INTERRUPT
0031                       *
0032                       *
0033    P0017    8622                ADD-    (ZERO),Q
0034    P0018    01B2                SNO     TEST-*-1
0035    P0019    D104                RAO-    4,I
0036    P001A    A011                AND-    LPMASK+15
0037    P001B    0D01      TEST      INQ     1
0038    P001C    4522                STQ-    (ZERO),I
0039    P001D    0852                TCQ     Q
0040    P001E    F105                ADQ-    5,I
0041    P001F    0141                SQZ     DV-*-1
0042    P0020    18EB                JMP*    LOOP
0043    P0021    E104      DV        LDQ-    4,I
0044    P0022    0FC1                ALS     1
0045    P0023    0F61                LRS     1
0046    P0024    3101      DIV       DVI-    1,I                    ANSWERS:
0047    P0025    6101                STA-    1,I                    3
0048    P0026    4500                STQ+    0,I                    4
        P0027    0000
0049    P0028    C103      EXIT      LDA-    3,I
0050    P0029    0500                IIN
0051    P002A    68D5                STA*    AVG
0052    P002B    54BA                RTJ-    (AVOLR)
0053    P002C    0400                EIN
0054    P002D    1CD2                JMP*    (AVG)
0055    P002E    0001                BZS     FLAG(1)
0056                                 END
```

```
0001                                    NAM       PGMB
0002                                    ENT       PGMB
0003                                    EXT       AVG, FINI
0004    P0000    0000       PGMB        0         0
0005    P0001    6817                   STA*      SA+1  }
0006    P0002    4814                   STQ*      SQ+1  }
0007    P0003    C0FF                   LDA-      I     }       SAVE REGISTERS
0008    P0004    680F                   STA*      SI+1  }
0009    P0005    0A08                   ENA       8                 8 NUMBERS
0010    P0006    E000                   LDQ       =XX               BUFFER ADDRESS X
        P0007    001B P
0011    P0008    5400  X                RTJ+      AVG               COMPUTE AVERAGE
        P0009    7FFF X
0012    P000A    6819                   STA*      ANS
0013    P000B    54F4                   RTJ-      ($F4)             PUNCH ANSWER
0014    P000C    0401                   NUM       $0401
0015    P000D    0000                   ADC       0
0016    P000E    0000                   NUM       0
0017    P000F    0003                   NUM       $0003
0018    P0010    0001                   NUM       1
0019    P0011    0023 P                 ADC       ANS
0020    P0012    C000       SI          LDA       =N0   }
        P0013    0000                                   }
0021    P0014    60FF                   STA-      I     }       RESTORE REGISTERS
0022    P0015    E000       SQ          LDQ       =N0   }
        P0016    0000                                   }
0023    P0017    C0000      SA          LDA       =N0   }
        P0018    0000
0024    P0019    1400  X                JMP+      FINI  }       RETURN TO AVG
        P001A    7FFF X
0025    P001B    0000       X           NUM       0, 1, 2, 3, 4, 5, 6, 7
        P001C    0001
        P001D    0002
        P001E    0003
        P001F    0004
        P0020    0005
        P0021    0006
        P0022    0007
0026    P0023    0001       ANS         BSS       ANS(1)
0027                                    END
```

ANSWER: 3

Another possible solution would be:

```
            NAM     AVG
            ENT     AVG
            EQU     AVOLA($BB), AVOLR($BA), ZERO($22)
            EQU     LPMASK($2), ONEBIT($23)
AVG         0       0
            IIN
            RTJ-    (AVOLA)
            NUM     5
            EIN
            LDA*    AVG
            STA-    3, I

            LDA-    I
            EOR-    ONEBIT+15
            STA-    I
            CLR     A
            STA-    4, I
            LDQ-    1, I
            INQ     -1
LOOP        ADD-    (I), Q
            SNO     TEST-*-1
            RAO-    4, I
            AND-    LPMASK+15
TEST        SQZ     DV-*-1
            INQ     -1
            JMP*    LOOP
DV          LDQ-    4, I
            ALS     1
            LRS     1
DIV         DVI-    1, I
            STA-    1, I
            STQ+    0, I
EXIT        LDA-    3, I
            IIN
            STA*    AVG
            LDA-    I
            AND-    LPMASK+15
            STA-    I
            RTJ-    (AVOLR)
            EIN
            JMP*    (AVG)
            END
```

The indirect bit on the contents of I causes proper addressing to be used to add up the buffer. Only 15 bits of the address are used for the direct addressing used to access the other volatile locations.

SOLUTION TO THREAD PROBLEM. (11.4.3.3)

Solution: Never.

The problem here is that the programmer thinks the request priority of 14 will override the running priority of 12, but this is not so. It is the driver's priority which must be considered, and the slow-equipment drivers usually run at 10. So, since the request is threaded as highest priority (14) on the queue for the logical unit (TTY, 4) and will be processed when the driver gets to run at its priority (10), the write will never be done. This is because the running program is hung in a loop at priority 12, waiting for an event which cannot occur (the thread word becoming zero) because the loop at 12 is locking out the driver.

Process programs usually run at 4, 5, and 6 (below the drivers); this would eliminate the problem in the example program. However, any looping like this at any priority is going to slow down a system by locking out lower priorities. For example, if many process programs were coded this way, they could almost completely lock out job processing (which runs at 0 and 1). It would be much better to code the write as follows, if it must run at 12:

```
          EQU       ADISP($EA)
           ≀
          RTJ-      ($F4)
          NUM       $OCED       FWRITE, RP=14, CP=13
          ADC       COMPL
          NUM       0
          NUM       $18FC
          NUM       35
          ADC       BUF
          SQP       OK-*-1
          JMP       REJ
OK        JMP-      (ADISP)
COMPL
```

SOLUTION TO MMPGM PROBLEM. (11.5.7)

Note that after the FWRITE is initiated (at P0017), the program return jumps to CORSUB and then schedules SYSPGM. CORSUB will run at the calling program's priority (4). Then, since SYSPGM is scheduled to run at 4 also, it will not begin until MMPGM is finished. (This is perfectly legal, as long as SYSPGM does not need any of the data in MMPGM and does not store anything in MMPGM.) At P001C a jump is then made to the dispatcher to await completion of the I/O. This all looks very good.

However, the I/O has been going on concurrently and the driver runs at a very high priority (usually 10). If by any chance it finishes the write and transfers control to the completion routine WROTE (at P001D) at priority 6 before the RTJ+ CORSUB and the schedule for SYSPGM are finished, the space MMPGM is in will be released. Surprise!

This is quite possible because other system programs at intermediate priorities (i.e., 7 and 9) could be locking out the MMPGM at 4, yet the driver at 10 would be plodding away at its write. Naturally the completion at 6 will be done (after the 7 and 8 are finished) before the priority drops back down to 4 to try to do the return jump to CORSUB and the schedule request, which, of course, aren't there any more.

The word on mass memory coding is: Be careful, and think!

A possible correction for the program would be:

```
            NAM         MMPGM
            ENT         MMPGM
            EXT*        REQREJ,IOERR
            EXT         CORSUB
            EXT         SYSPGM
ADISP       EQU         ADISP($EA)
MMPGM       NUM         $C8FE
            STA*        REL+2
            JMP*        WRITE
MSGBUF      ALF         *,MASS MEMORY EXAMPLE*
WRITE       FWRITE      $FC,WROTE-*+1,MSGBUF-*+5,10,A,5,6,I,X
            SQP         REQOK-*-1
            RTJ         REQREJ
REQOK       RTJ+        CORSUB
            JMP-        (ADISP)
WROTE       SQP         2
            RTJ         IOERR
            SCHDLE      REL,4,X
            SQP         2
            RTJ         REQREJ
            SCHDLE      (SYSPGM),4,0
            SQP         2
            RTJ         REQREJ
            JMP-        (ADISP)
REL         RELEAS      0,T,0
            END         MMPGM
```

In this solution, if the completion routine is entered before the RTJ+ to CORSUB is finished, REL and SYSPGM will be put on the scheduler stack to be executed after the completion exit to the dispatcher allows MMPGM to be picked up from the interrupt stack.

Another possible correction to the program, perhaps better, would be simply to change the completion priority in the FWRITE request from 6 to 3. That would insure that any priority 4 work would be finished before the release is executed. However, this would cause the space MMPGM is in to be tied up until SYSPGM is finished, which was not the intent.

APPENDIX H

```
                                                      1700
********PP********                                    ADDRESSING EXAMPLES
0001                 *P                               ASSEMBLED UNDER MSOS 2.0
0002                 *ASSEM                           OCT. 1968
********OP********
********UD********
0003  P0000 0000             NAM   L
********OP********
********UD********
0004  P0001 0000              NAM    DAN
0005        0000              ORG         0           SET  ABSOLUTE  ADDRESS
0006   0000 0B00              NOP
0007   0001 7FFF X   LOWCOR   0           DPNDNT      EXTERNAL SYMBOL
0008        00FE              ORG         $FE            RE-SET  ABSOLUTE  ADDRESS
0009   00FE 00BD P   TEMP1    0           ANYWHR
0010   00FF 0000     INDEX2   0           0
0011   0100 0000     ABSRNG   0           0
0012        0002  P           ORG*
0013  P0002 0000     RELOW    0           0
0014                 *                                1700  ASSEMBLY  EXAMPLES
0015                 *                                ********STORAGE  REFERENCE********
0016                 *                                         **GROUP  1**
0017                 *         EVALUATION OF ADDRESS FIELD RESULTS IN DESIRED OPERAND ADDRESS
0018                 *        *ABSOLUTE*
0019                 *LABEL*  *OPN*       *ADDRESS*   *DESCRIPTION*           *BA*        *EFA*
0020  P0003 C0FE               LDA-        TEMP1        NO  INDEXING            TEMP1   TEMP1
0021  P0004 C2FE               LDA-        TEMP1,Q      Q  INDEXING             TEMP1   TEMP1+(Q)
0022  P0005 C1FE               LDA-        TEMP1,I      I  INDEXING             TEMP1   TEMP1+(00FF)
0023  P0006 C3FE               LDA-        TEMP1,B      DOUBLE INDEXING         TEMP1   TEMP1+(Q)+(00F
0024  P0007 C020               LDA-        $20          NUMERIC HEX EXAMPLE     20 HEX  20 HEX
0025  P0008 C014               LDA-        20           NUMERIC DEC EXAMPLE     20 HEX  20 DEC
********FX********
0026  P0009 C000               LDA-        ABSRNG       DELTA OUT OF RANGE
********PL********
0027  P000A C002               LDA-        RELOW        PROGRAM RELOCATABLE ADDRESS
********RL********
0028  P000B C010               LDA-        BLOCK6       DATA RELOCATABLE ADDRESS
********RL********
0029  P000C C000               LDA-        BLOCK3       COMMON RELOCATABLE
********UD********
0030  P000D C000               LDA-        UNDEFINED    UNDEFINED SYMBOL
```

```
0032                 *      *RELATIVE*
0033                 *LABEL*  *OPN*    *ADDRESS*          *DESCRIPTION*        *BA*      *EFA*
0034  P000E  C822    BAKREL   LDA*      RELADR            DELTA=RELADR+(-*)    RELADR    RELADR
0035  P000F  CA21             LDA*      RELADR,Q          DELTA=RELADR+(-*)    RELADR    RELADR+(Q)
0036  P0010  CB20             LDA*      RELADR,B          DELTA=RELADR+(-*)    RELADR    RELADR+(Q)+(00FF)
0037  P0011  C8FC             LDA*      BAKREL            DELTA=BAKREL+(-*)    BAKREL    BAKREL
0038  P0012  C8FD             LDA*      *-2               DELTA+*-2+(-*)       *-2       *-2
0039  P0013  181D             JMP*      RELADR            JUMP ADDRESS=EFA     RELADR      RELADR
********RL********
0040  P0014  C017             LDA*      TAGLRM-BAKREL     FORM  RELATIVE  ADDRESS
********RL********
0041  P0015  C010             LDA*      $10               ABSOLUTE  ADDRESS
********RL********
0042  P0016  C0000            LDA*      ABSRNG            ABSOLUTE  ADDRESS

********FX********
0043  P0017  C0A6             LDA*      ANYWHR            DELTA  OUT  OF  RANGE


0045                 *                              ****CONTINUATION OF STORAGE REFERENCE****
0046                 *                                          ****GROUP 1****
0047                 *   *LONG  RELATIVE*
0048                 *LABEL*  *OPN      *ADDRESS*          *DESCRIPTION*        *BA*      *EFA*
0049  P0018  C800             LDA       RELADR            M=RELADR+(-*.+1)     RELADR    RELADR
      P0019  0017
0050  P001A  C800             LDA       ANYWHR            M=16  BIT  REL ADDRESS  ANYWHR  ANYWHR
      P001B  00A2
0051  P001C  CA00             LDA       ANYWHR,Q          Q  INDEXING          ANYWHR    ANYWHR+(Q)
      P001D  00A0
0052  P001E  5800             RTJ       ANYWHR            JUMP ADDRESS=EFA     ANYWHR    ANYWHR
      P001F  009E
0053  P0020  C400             LDA       TEMP1             ABSOLUTE ADDRESS, ASSEMBLED AS STORAGE
      P0021  00FE
0054  P0022  C400             LDA       $500              ASSEMBLED AS STORAGE MODE
      P0023  0500
0055  P0024  C800             LDA*      *                 WORD  ONE  OF  LONG  RELATIVE
0056  P0025  0000    TAGLRM   0         0                 WORD  TWO  OF  LONG  RELATIVE
```

```
0058                  *       *STORAGE*
0059                  *LABEL*  *OPN*      *ADDRESS*        *DESCRIPTION*           *BA*        *EFA*
0060 P0026 C400       BACIND    LDA+      ANYWHR           2ND WORD=ANYWHR         ANYWHR      ANYWHR
     P0027 00BD P
0061 P0028 C600                 LDA+      ANYWHR,Q         Q INDEXING              ANYWHR      ANYWHR+(Q)
     P0029 00BD P
0062 P002A C700       MEMADR    LDA+      $500,B           NUMERIC EXAMPLE         $500        $500+(Q)+(00FF)
     P002B 0500
0063 P002C C400                 LDA-      (0)              WORD ONE OF STORAGE MODE
0064 P002D 0000       TAGSM     0         0                WORD TWO OF STORAGE MODE
0065 P002E 5400                 RTJ+      ANYWHR           JUMP ADDRESS=EFA        ANYWHR        ANYWHR
     P002F 00BD P

0067                  *
0068                  *
0069                  *       EVALUATION OF ADDRESS FIELD RESULTS IN DESIRED ADDRESS OF OPERAND ADDRESS
0070                  *       *INDIRECT*
0071                  *LABEL*  *OPN*      *ADDRESS*        *DESCRIPTION*            *BA*        *EFA*
0072 P0030 C4FE       RELADR    LDA-      (TEMP1)          DELTA=TEMP1             (TEMP1)     (TEMP1)
0073                  *         LDA-      (TEMP1),Q        DELTA=TEMP1             (TEMP1)     (TEMP1)+(Q)
0074                  *         LDA-      (TEMP1),B        DELTA=TEMP1             (TEMP1)     (TEMP1)+(Q)+(00FF)
0075 P0031 C4FF                 LDA-      (TEMP1+1)        DELTA=TEMP1+1           (TEMP1+1)   (TEMP1+1)
0076 P0032 C420                 LDA-      ($20)            DELTA=$20               ($20)       ($20)
0077 P0033 14FE                 JMP-      (TEMP1)          JUMP ADDRESS=EFA        (TEMP1)       ANYWHR

0079                  *       *STORAGE INDIRECT*
0080                  *LABEL*  *OPN*      *ADDRESS*        *DESCRIPTION*            *BA*        *EFA*
0081 P0034 C600       RELIND    LDA+      (BACIND+1),Q     2ND WORD=BACIND+1       (BACIND+1)  ANYWHR+(Q)
     P0035 8027  P
0082 P0036 C700                 LDA+      (MEMADR+1),B     2ND WORD=MEMADR+1     (MEMADR+1)    $500+(Q)+(00FF)
     P0037 802B  P
0083 P0038 C400                 LDA+      ($700)           NUMERIC EXAMPLE         ($700)      ($700)
     P0039 8700
0084 P003A 5400                 RTJ+      (BACIND+1)       JUMP ADDRESS=EFA        (BACIND+1)    ANYWHR
     P003B 8027  P

0086                  *       *RELATIVE INDIRECT*
0087                  *LABEL*  *OPN       *ADDRESS*        *DESCRIPTION*            *BA*        *EFA*
0088 P003C CCEA                 LDA*      (BACIND+1)       DELTA=BACIND+1+(-*)     (BACIND+1)    ANYWHR
0089 P003D CDED                 LDA*      (MEMADR+1),I     DELTA=MEMADR+1+(-*)   (MEMADR+1)      $500+(00FF)
0090 P003F CFF6                 LDA*      (RELIND+1),B     DELTA=RELIND+1+(-*)     (RELIND+1) ANYWHR+(Q)+(00FF)
```

```
0092                   *    *LONG RELATIVE INDIRECT*
0093                   *LABEL*  *OPN*    *ADDRESS*     *DESCRIPTION*            *BA*          *EFA*
0094  P003F CC00                LDA      (BACIND+1)    M=BACKIND+(-*)+1         (BACIND+1)    ANYWHR
      P0040 FFE6
0095  P0041 CE00                LDA      (ANYWHR),Q    16 BIT REL ADDRESS       (ANYWHR)      (ANYWHR)+(Q)
      P0042 007B
0096  P0043 C400                LDA      ($700)        ASSEMBLED AS STORAGE INDIRECT
      P0044 8700

0098                   *                   ****CONTINUATION OF STORAGE REFERENCE****
0099                   *                                 ****GROUP 3****
0100                   *             EVALUATION OF ADDRESS FIELD RESULTS IN DESIRED OPERAND
0101                   *                 N = NUMERIC        X = ADDRESS        A = ALPHANUMERIC

0103                   *        *CONSTANT*
0104                   *LABEL*  *OPN*    *ADDRESS*     *DESCRIPTION*        *BA*          *EFA*
0105  P0045 C000                LDA      =N1000        NUMERIC EXAMPLE DEC  P+1           P+1
      P0046 03E8
0106  P0047 C000                LDA      =N$1000       NUMERIC EXAMPLE HEX  P+1           P+1
      P0048 1000
0107  P0049 C000                LDA      =N-$1000      NUMERIC EXAMPLE NEG  P+1           P+1
      P004A EFFF
0108  P004B C000                LDA      =XBACIND      ADDRESS EXAMPLE      P+1           P+1
      P004C 0026  P
0109  P004D C000                LDA      =X-BACIND     ADDRESS EXAMPLE NEG  P+1           P+1
      P004E 7FD9-P
0110  P004F C000                LDA      =ABC          ALPHA EXAMPLE        P+1           P+1
      P0050 4243
0111  P0051 C200                LDA      =N$1000,Q     RA=(P+1)  *INDEXING*  $1000        OPERAND
      P0052 1000
0112  P0053 C000                LDA-     0             WORD ONE OF CONSTANT MODE
0113  P0054 0000       TAGCM    0        0             WORD TWO OF CONSTANT
********EX********
0114  P0055 6400                STA      =N$0          ASSEMBLER WILL NOT ALLOW
      P0056 0000
0115  P0057 C000                LDA      =X-4          NUMERIC WITH ADDRESS INDICATOR
      P0058 7FFB
0116  P0059 C000                LDA      =X(BACIND)    SET INDIRECT BIT ON ADDRESS VALUE
      P005A 8026  P
```

```
0118                  *                       ********REGISTER TRANSFER********
0119                  *       EVALUATION OF ADDRESS FIELD RESULTS IN DESIRED DESTINATION REGISTER
0120                  *LABEL* *OPN*   *ADDRESS*               *DESCRIPTION*
0121  P005B 0847              CLR      A,Q,M        CLEAR A  Q  AND  M  REGISTERS
0122  P005C 080C              TRM      A            TRANSFER  M  TO  A
0123  P005D 0811              TRQ      M            TRANSFER  Q  TO  M
0124  P005E 0823              TRA      Q,M          TRANSFER  A  TO  Q  AND  M
0125  P005F 081C              TRB      A            TRANSFER INCLUSIVE OR OF M  Q  TO  A
0126  P0060 0801              SET      M            SET  M  TO ALL ONES
0127  P0061 0854              TCQ      A            TRANSFER THE COMPLEMENT OF  Q  TO  A
0128  P0062 0834              AAQ      A            TRANSFER SUM OF  A  Q  TO  A
0129  P0063 086A              EAM      Q            TRANSFER EXCLUSIVE OR OF  A  M  TO  Q
0130  P0064 08B2              LAQ      Q            TRANSFER LOGICAL AND OF  A  Q  TO  Q
0131  P0065 08F4              CAQ      A            TRANSFER COMPLEMENT OF LOGICAL AND OF  A  Q  TO  A
********EX********
0132  P0066 0820              TRA      X            ILLEGAL DESTINATION REGISTER

0134                  *                         ********SKIPS********
0135                  *       EVALUATION OF ADDRESS FIELD RESULTS IN DESIRED SKIP COUNT
0136                  *LABEL* *OPN*   *ADDRESS*               *DESCRIPTION*
0137  P0067 0105              SAZ      5            IF(A) IS 0 SKIP FORWARD 6 PLACES ++1+SKIPCOUNT
0138  P0068 0156              SQN      SKIPAD-*-1   IF(Q) NEG SKIP TO SKIPAD (NON ABSOLUTE METHOD)
0139  P0069 0115              SAN      SKIPAD       SKIP COUNT OUT OF RANGE
********EX********
0140  P006A 0120              SAP      $10          SKIP COUNT OUT OF RANGE
********EX********
0141  P006B 013A              SAM      -5           NEG SKIP COUNT

0143                  *                        ********SHIFTS********
0144                  *       EVALUATION OF ADDRESS FIELD RESULTS IN DESIRED SHIFT COUNT
0145                  *LABEL* *OPN*   *ADDRESS*               *DESCRIPTION*
0146  P006C 0FAC              QLS      12           LEFT SHIFT  Q  12 DECIMAL POSITIONS
0147  P006D 0FD2              ALS      $12          LEFT SHIFT  A  12 HEX POSITIONS
0148  P006E 0F7F              LRS      31           RIGHT SHIFT Q/A 31 DECIMAL POSITIONS  (31 = MAX)
0149  P006F 0B0A      SKIPAD  NOP      10           WAIT XX MICROSECONDS
0150  P0070 0F53              ARS      $33          SHIFT COUNT GREATER THAN $31
0151  P0071 0F5F              ARS      DELTA        SYMBOLIC SHIFT COUNT
0152  P0072 0FDA              ALS      -5           NEG SHIFT COUNT
```

```
0154                    *
0155                    *  EVALUATION OF ADDRESS FIELD RESULTS IN VARIOUS PORTIONS OF ASSEMBLED INSTS
0156                    *LABEL*  *OPN*    *ADDRESS*          *DESCRIPTION*
0157  P0073 0A75                 ENA      $75          ENTER A WITH VALUE SPECIFIED IN ADDRESS FIELD
0158  P0074 0CEA                 ENQ      -$15         NEG EXAMPLE
0159  P0075 0A7F                 ENA      127          DEC EXAMPLE (MAX VALUE)
********EX********
0160  P0076 0A80                 ENA      128          NEG. NUMERIC VALUE
********EX********
0161  P0077 0A01                 ENA      257          OUT OF RANGE NUMERIC VALUE
0162  P0078 0A1F                 ENA      DELTA        POS. SYMBOLIC VALUE
0163  P0079 0AE0                 ENA      -DELTA       NEG. SYMBOLIC VALUE
********EX********
0164  P007A 0AFF                 ENA      INDEX2       NEG. SYMBOLIC VALUE
********EX********
0165  P007B 0A02                 ENA      RELOW        PROGRAM RELOCATABLE VALUE
********EX********
0166  P007C 0A10                 ENA      BLOCK6       DATA RELOCATABLE VALUE
********EX********
0167  P007D 0A00                 ENA      BLOCK3       COMMON RELOCATABLE VALUE
0168  P007E 09FE                 INA      -1           INCREASE A BY THE VALUE SPECIFIED IN ADDRESS FD
0169  P007F 0D25                 INQ      $25          INCREASE Q
0170  P0080 0206                 INP      REJCTA-*-1   INPUT TO A REJECT ADDRESS-P+1+DELTA
********EX********
0171  P0081 0287                 INP      REJCTA       DELT OUT OF RANGE
0172  P0082 0400                 EIN      0            ENABLE INTERRUPT SYSTEM
0173  P0083 0500                 IIN      0            INHIBIT INTERRUPT SYSTEM
0174  P0084 0E04                 EXI      4            EXIT INTERRUPT ADDRESS FIELD = INTERRUPT
0175  P0085 0600                 SPB      0            SET PROTECT BIT (Q) = ADDRESS
0176  P0086 0700                 CPB      0            CLEAR PROTECT BIT (Q) = ADDRESS
0177  P0087 0000        REJCTA   SLS      0            STOP IF STOP KEY SET
0178        001F                 EQU      DELTA($1F)     EQUATE SYMBOLIC VALUE FOR ABOVE USE
```

```
0180                  *                                        ********CLASS 2 PSEUDOS********
0181                  *    *ADC* ADDRESS CONSTANT PSEUDO
0182                  *          THE ADDRESS EXPRESSIONS IN SUBFIELD ARE ASSEMBLED INTO CONSECUTIVE CELL
0183                  *          LOCATIONS.  IF ADDRESS EXPRESSION IS ENCLOSED IN PARENTHESIS THE ADDRESS
0184                  *          BECOMES INDIRECT.
0185                  *LABEL*  *OPN*     *ADDRESS*
0186   P0088 0030  P  ADLIST     ADC       RELADR, BACIND, (MEMADR), -RELADR
       P0089 0026  P
       P008A 802A  P
       P008B 7FCF-P
0187   P008C 7FF0                0         -$F              TREATED AS ONE WORD ADC

0189                  *    *NUM* NUMERIC CONSTANT PSEUDO
0190                  *          THE NUMERIC EXPRESSIONS IN SUBFIELD ARE ASSEMBLED INTO CONSECUTIVE CELL
0191                  *          LOCATIONS.  THE CONSTANTS CAN BE EITHER DECIMAL OR HEX VALUES.
0192                  *LABEL*    *OPN*   *ADDRESS*
0193   P008D 1000     CONLST     NUM       $1000, 1000, -$1000, -100
       P008E 03E8
       P008F EFFF
       P0090 FF9B

0195                  *    *ALF* ALPHANUMERIC MESSAGE PSEUDO
0196                  *          THE ADDRESS FIELD CONTAINS THE NUMBER OF CELLS TO BE RESERVED FOR THE
0197                  *          REMAINING CHARACTERS IN FIELD.
0198                  *LABEL*  *OPN*     *ADDRESS*
```

```
0200                      *           *****CONTINUATION OF CLASS 2 PSEUDOS*****
0201                      *    *ENT* ENTRY PSEUDO
0202                      *          THIS PSEUDO WILL CAUSE A BINARY OUTPUT WHICH WILL ALLOW EXTERNAL
0203                      *          SYMBOLS OF OTHER PROGRAMS TO BE DEFINED AT *LOAD TIME*
0204                      *         *OPN*     *ADDRESS*
0205                                 ENT       START,BACIND

0207                      *    *EXT* EXTERNAL PSEUDO
0208                      *          THIS PSEUDO WILL ALLOW SYMBOLIC VALUES UNDEFINED IN THE INDEPENDENT
0209                      *          PROGRAM TO BE MATCHED WITH ENTRY VALUES AND DEFINED AT LOAD TIME.
0210                      *         *OPN*     *ADDRESS*         *DESCRIPTION*
0211                                 EXT       DPNDNT            DPNDNT WILL BE MATCHED WITH ENTRY POINT
0212                      *                                          AT LOAD TIME.
0213  P0091  5400  X                 RTJ       DPNDNT            DPNDNT IS UNDEFINED IN CURRENT ROUTINE
      P0092  0001  X
0214  P0093  0B00                    NOP
0215  P0094  5400  X                 RTJ       DPNDNT            ASSEMBLED AS STORAGE
      P0095  0092  X
0216  P0096  0B00                    NOP
0217  P0097  5401                    RTJ-      (LOWCOR)          INDIRECT LINKING
0218  P0098  0B00                    NOP
********EX********
0219  P0099  5400  X                 RTJ*      DPNDNT
0220  P009A  0B00                    NOP
********EX********
0221  P009B  5000  X                 RTJ-      DPNDNT
```

```
0223                   *                                       ****CLASS 3 PSEUDOS****
0224                   *   *EQU*  EQUATE PSEUDO
0225                   *          THIS PSEUDO WILL CAUSE A SYMBOLIC VALUE TO BE EQUATED TO ANOTHER
0226                   *          SYMBOLIC VALUE OR TO A NUMERIC VALUE AND PLACED IN SYMBOL TABLE.
0227                   *         *OPN*      *ADDRESS*
0228         0100                 EQU        START($100), HERE(*)
             009C P

0230                   *   *BSS*  BLOCK STORE PSEUDO
0231                   *          THIS PSEUDO WILL CAUSE A RESERVATION OF THE NUMBER OF CELLS SPECIFIED
0232                              BY THE VALUE IN THE ADDRESS FIELD. THE CONTENTS OF THESE CELLS WILL BE
0233                              UNCHANGED AT LOAD TIME.
0234                   *LABEL*   *OPN*      *ADDRESS*         *DESCRIPTION*
0235    P009C 0B00                NOP        0                 INDICATOR TO SHOW CURRENT ADDRESS
0236    P009D 0010                BSS        BLOCK8($10)
0237    P00AD 0B00                NOP        0                 INDICATOR TO SHOW CURRENT ADDRESS

0239                   *   *BZS*  BLOCK ZERO STORE PSEUDO
0240                   *          SAME AS BSS EXCEPT CELLS WILL BE SET TO ZERO AT LOAD TIME.
0241                   *LABEL*   *OPN*      *ADDRESS*         *DESCRIPTION*
0242    P00AE 0B00                NOP        0                 INDICATOR TO SHOW CURRENT ADDRESS
0243    P00AF 0005     BLOCK2     BZS        BLOCK9($5)
0244    P00B4 0B00                NOP        0                 INDICATOR TO SHOW CURRENT ADDRESS

0246                   *   *COM*  COMMON STORAGE PSEUDO
0247                   *          THE NAME OF THE BLOCKS AND THE SIZE ARE DEFINED IN THE ADDRESS FIELD
0248                   *          OF PSEUDO.  THE STORAGE AREA WILL BE ASSIGNED TO THE AREA OF THE
0249                   *          LOADER AT LOAD TIME.
0250                   *         *OPN*      *ADDRESS*         *DESCRIPTION*
0251         0000 C               COM        BLOCK3($30), BLOCK4($100)      BLOCK4=BLOCK3+$30
             0030 C

0253                   *   *DAT*  DATA STORAGE PSEUDO
0254                   *          THE METHOD OF RESERVATION IS THE SAME AS COM EXCEPT THE AREA CAN
0255                   *          BE PRESET.
0256                   *         *OPN*      *ADDRESS*         *DESCRIPTION*
0257         0000 D               DAT        BLOCK5($10), BLOCK6($20*$20)
             0010 D
0258    P00B5 0B00                NOP        0                 INDICATOR TO SHOW CURRENT ADDRESS
```

```
0260                   *                     METHODS OF PRESETTING DATA IN DATA AREA
0261                   *                   ALSO SHOWN IS ILLEGAL USE OF COMM AREA

0263         0000 D              ORG       BLOCK5       PRESET TO DATA AREA
0264    D0000 0088 P             ADC       ADLIST,MEMADR,CONLST,HERE,(WORD2A)
        D0001 002A P
        D0002 008D P
        D0003 009C P
********UD********
        D0004 8000
0265    D0005 0001               NUM       1,2,3,4,5
        D0006 0002
        D0007 0003
        D0008 0004
        D0009 0005
0266          00B6 P             ORG*      0                RESET TO NORMAL COUNTER
0267    P00B6 0B00               NOP       0                INDICATOR TO SHOW CURRENT ADDRESS
********RL********
0268          0000               ORG       BLOCK3           SET COUNTER TO COMMON AREA
0269    P00B7 0001               NUM       1,2,3,4,5        ILLEGAL TO SET DATA IN COMMON
        P00B8 0002
        P00B9 0003
        P00BA 0004
        P00BB 0005

0271                   *                             ********CLASS 1 PSEUDOS********
0272                   *    NAM  NAME PSEUDO
0273                   *         THE NUMERIC VALUE IN LABEL FIELD WILL SET PROGRAM COUNTER TO AN
0274                   *         ABSOLUTE VALUE. THE ADDRESS FIELD CONTAINS THE PROGRAM NAME.
0275                   *         ONLY ONE NAM ALLOWED PER PROGRAM. NAM IS USED TO IDENTIFY INDE-
0276                   *         PENDENT PROGRAMS.

0278                   *    ORG PROGRAM COUNTER CONTROL PSEUDO
0279                   *         COUNTER. IF NUMERIC IS USED PROGRAM WILL BE ASSEMBLED ABSOLUTE.
```

```
0281                  *    ORG*  RETURN PROGRAM COUNTER PSEUDO
0282                  *          USED TO RETURN COUNTER TO NORMAL VALUE
0283                  *         *OPN*       *ADDRESS*      *DESCRIPTION*
0284                             ORG*
0285   P00BC 0B00                NOP          0              INDICATOR TO SHOW CURRENT ADDRESS

0287         00BD P              EQU         ANYWHR(*)

0289                  *   *END*  END PSEUDO
0290                  *    MUST BE LAST CARD OF EACH PROGRAM, SYMBOL IN ADDRESS FIELD IS THE
0291                  *    ADDRESS CONTROL WILL BE TRANSFERRED TO AT LOAD TIME.
0292                  *   *OPN*                 *ADDRESS*
0293                  *
0294                  *
0295                       END                  START
```

```
I            00FF     LOWCOR    0001    TEMP1      00FE     INDEX2    00FF     ABSRNG     0100
RELOW        0002P    BAKREL    000EP   TAGLRM     0025P    BACIND    0026P    MEMADR     002AP
TAGSM        0002DP   RELADR    0030P   RELIND     0034P    TAGCM     0054P    SKIPAD     006FP
REJCTA       0087P    DELTA     001F    ADLIST     0088P    CONLST    008DP    START      0100
HERE         009CP    BLOCK8    009DP   BLOCK9     00AFP    BLOCK3    0000C    BLOCK4     0030C
BLOCK5       0000D    BLOCK6    0010D   ANYWHR     00BDP    DPNDNT    0095X

030  ERRORS
```

APPENDIX I

APPENDIX I

COMMUNICATION REGION

The communication region is the area of core below FF_{16}. It can be addressed directly by a one-word instruction. Contents are defined by the following table. All locations are protected except as noted. EQU names are noted also.

	Location	Contents	HEX Equivalent
	0 & 1	Reserved for the system	
LPMASK →	2	0000000000000000	0
	3	0000000000000001	1
	4	0000000000000011	3
	5	0000000000000111	7
	6	0000000000001111	F
	7	0000000000011111	1F
	8	0000000000111111	3F
	9	0000000001111111	7F
	A	0000000011111111	FF
	B	0000000111111111	1FF
	C	0000001111111111	3FF
	D	0000011111111111	7FF
	E	0000111111111111	FFF
	F	0001111111111111	1FFF
	10	0011111111111111	3FFF
	11	0111111111111111	7FFF
NZERO →	12	1111111111111111	FFFF
	13	1111111111111110	FFFE
	14	1111111111111100	FFFC
	15	1111111111111000	FFF8
	16	1111111111110000	FFF0
	17	1111111111100000	FFE0
	18	1111111111000000	FFC0
	19	1111111110000000	FF80
	1A	1111111100000000	FF00
	1B	1111111000000000	FE00
	1C	1111110000000000	FC00
	1D	1111100000000000	F800
	1E	1111000000000000	F000
	1F	1110000000000000	E000
	20	1100000000000000	C000
	21	1000000000000000	8000
ZERO →	22	0000000000000000	0000

	Location	Contents	HEX Equivalent
ONEBIT →	23	0000000000000001	0001
	24	0000000000000010	0002
	25	0000000000000100	0004
	26	0000000000001000	0008
	27	0000000000010000	0010
	28	0000000000100000	0020
	29	0000000001000000	0040
	2A	0000000010000000	0080
	2B	0000000100000000	0100
	2C	0000001000000000	0200
	2D	0000010000000000	0400
	2E	0000100000000000	0800
	2F	0001000000000000	1000
	30	0010000000000000	2000
	31	0100000000000000	4000
	32	1000000000000000	8000
ZROBIT →	33	1111111111111110	FFFE
	34	1111111111111101	FFFD
	35	1111111111111011	FFFB
	36	1111111111110111	FFF7
	37	1111111111101111	FFEF
	38	1111111111011111	FFDF
	39	1111111110111111	FFBF
	3A	1111111101111111	FF7F
	3B	1111111011111111	FEFF
	3C	1111110111111111	FDFF
	3D	1111101111111111	FBFF
	3E	1111011111111111	F7FF
	3F	1110111111111111	EFFF
	40	1101111111111111	DFFF
	41	1011111111111111	BFFF
	42	0111111111111111	7FFF
	43	5	
	44	6	
	45	9	
	46	A_{16}	
	47 through B2	Reserved for process	
	B3	Logical unit number of scratch unit	
	B4	Top of thread of entries in schedule stack	
	B5	Address of FNR	
	B6	Address of COMPRQ	
	B7	Address of mask table	
	B8	Address of top of interrupt stack	
	B9	Address of request exit	

Location	Contents	HEX Equivalent
BA	Address of volatile storage release routine - VOLR	
BB	Address of volatile storage assignment routine - VOLA	
BC	Address of absolutizing routine for logical unit number	
BD	Address of S absolutizing routine	
BE	Address of C absolutizing routine	
BF	Address of N absolutizing routine	
C0	Most significant bits of first scratch area sector number	
C1	Least significant bits of first scratch area sector number	
C2	Logical unit number of the library unit	
C3	Most significant bits of sector number of first program library directory block	
C4	Least significant bits of sector number of first program library directory block	
C5 through E3	Reserved for FORTRAN (unprotected)	
E4	Used for load and go (unprotected)	
E5	Address of timer handler	
E6	Length of system library directory	
E7	Index to first mass storage entry in the system library directory	
E8	Countdown register	
E9	Real time clock	
EA	Address of dispatcher	
EB	Address of system library directory	
EC	Temporary highest unprotected location + 1	
ED	Temporary lowest unprotected location - 1	
EE	Used by job processor for returns from loader, etc.	
EF	Current priority level - PRLVL	
F0	Address of first available volatile storage	
F1	Length of table of presets	
F2	Address of table of presets	
F3	Address of breakpoint program when in core (unprotected)	
F4	Address of entry for system requests (unprotected)	
F5	Highest core location - MAXCOR	
F6	Highest unprotected location + 1	
F7	Lowest unprotected location - 1	
F8	Address of internal interrupt processor	
F9	Logical unit number of standard input device	
FA	Logical unit number of standard binary output device	
FB	Logical unit number of standard print output device	
FC	Logical unit number of output comment device	
FD	Logical unit number of input comment device	
FE	Address of the common interrupt handler	
FF	Memory index (unprotected) - I register	

INDEX

I

INDEX

INDEX (CONT)

INDEX (CONT)

CONTROL DATA
CORPORATION