DATA GENERAL
CORPORATION
Southboro,
Massachusetts 01772
(617) 485-9100

INTRODUCTION

TO

PROGRAMMING

THE NOVA COMPUTERS

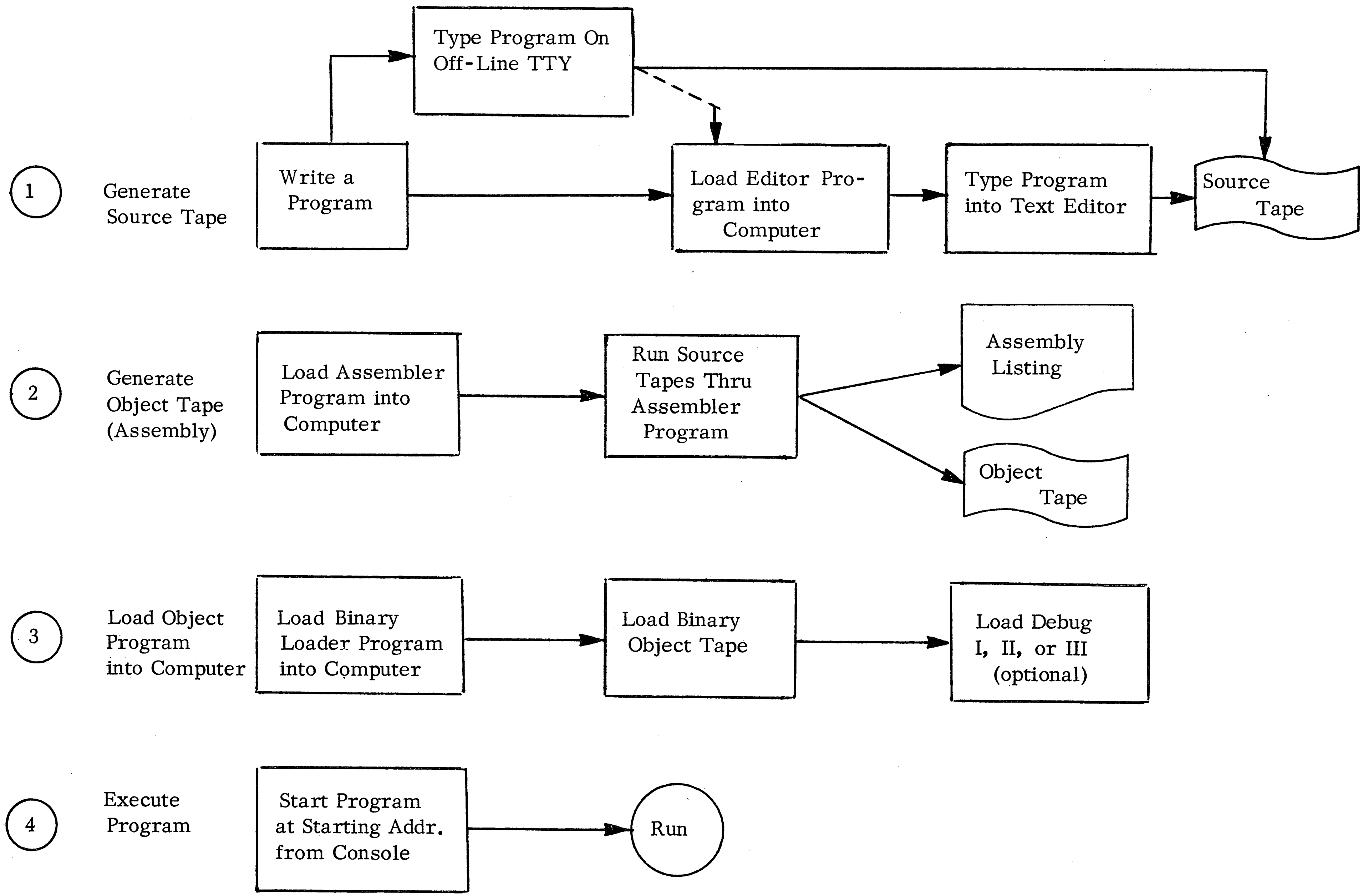
A general description of how to write a program in Nova assembly language, edit the source program, assemble the source program into an object program, debug the object program, and load and run the object program.

093-000067-01

This manual has been adapted from the notebook used in the Data General Programming Course. It contains an overview of programming of the Nova computers for programmers with limited or intermediate programming experience.

It is intended primarily as a general presentation of how the programmer writes, edits, assembles, loads, debugs, and runs programs on Nova line computers, with limited descriptions of the basic DGC programs available for these purposes.

A limited abstract/bibliography of some of the DGC publications describing DGC system programs mentioned in this manual is contained in Appendix B.



Type Program On Off-Line TTY
1
Generate Source Tape
Write a Program
Load Editor Program into Computer
Type Program into Text Editor
Source Tape
2
Generate Object Tape (Assembly)
Load Assembler Program into Computer
Run Source Tapes Thru Assembler Program
Assembly Listing
Object Tape
3
Load Object Program into Computer
Load Binary Loader Program into Computer
Load Binary Object Tape
Load Debug I, II, or III (optional)
4
Execute Program
Start Program at Starting Addr. from Console
Run

TABLE OF CONTENTS

FOREWORD i

CHART OF NOVA PROGRAMMING ii

CHAPTER I - INTRODUCTION 1-1

CHAPTER 2 - CENTRAL PROCESSING UNIT INSTRUCTIONS

Memory Reference Instructions (MRI) 2-1
Move Data MRI's 2-3
Modify Memory MRI's 2-4
Jump MRI's 2-4
Arithmetic and Logical Instructions (ALC) 2-6
Arithmetic and Logical Functions 2-7
Carry Field 2-8
Shift Field 2-9
Skip Field 2-10
Load/No Load Field 2-10
Input/Output Instructions (I/O) 2-11
Data Transfer 2-11
Control Field 2-12
Special Functions 2-13

CHAPTER 3 - BASIC ASSEMBLER

Label 3-1
Opcode 3-1
Operand 3-1
Comment 3-2
Program Format 3-2
Special Characters 3-2
Assembler Pseudo-ops 3-4
.LOC 3-4
.BLK 3-5
.RDX 3-5
.TXT 3-6
.TXTM 3-6
Source Program Termination Pseudo-ops 3-7
.END 3-7
.EOT 3-7

CHAPTER 3 - BASIC ASSEMBLER (Coninuted)

Symbol Table Pseudo-ops 3-7
.DUSR 3-7
.DMR 3-8
.DMRA 3-9
.DALC. 3-9
.DIO 3-10
.DIOA 3-10
.DIAC 3-10
Assembler Operating Procedures 3-11

CHAPTER 4 - TEXT EDITOR 4-1

CHAPTER 5 - CONSOLE OPERATION

Power Switch 5-1
Indicators 5-1
Switches 5-2

CHAPTER 6 - PROGRAM LOADER

Bootstrap Loader 6-1
Binary Loader 6-2

CHAPTER 7 - NUMERIC DEBUGGER

Debug I 7-1
Debug III 7-1
Debug II 7-1
Examining and Modifying Registers 7-1
Searching Memory 7-2
Breakpoints 7-3
Punching an Object Tape 7-4
Calculations using Debug II 7-5

CHAPTER 8 - I/O DEVICE HANDLING

Program Interrupts 8-1
Interrupt Service Routines 8-3
Power Monitor and Auto-restart 8-5
Data Channel 8-5

CHAPTER 9 - EXTENDED SOFTWARE

Relocatability 9-1
Interprogram Communication 9-4
Assembler Extensions 9-8
Floating Point Numbers 9-8
Double Precision Numbers 9-8
Bit Boundary Alignment 9-9
Conditional Assembly 9-10
Symbolic Debugger 9-11

CHAPTER 10 - BYTE MANIPULATION 10-1

CHAPTER 11 - PROGRAMMING TRICKS 11-1

APPENDIX A - SAMPLE PROGRAMS

APPENDIX B - BIBLIOGRAPHY

APPENDIX C - PSEUDO-OPS

APPENDIX D - INSTRUCTION MNEMONICS AND TIMING

APPENDIX E - IN-OUT CODES

APPENDIX F - ASSEMBLY ERROR FLAGS

INDEX

CHAPTER 1

INTRODUCTION

The Nova family uses 16-bit words; the bits are labeled from left to right, Ø through 15. The core memory (core storage) capacity is a maximum of 32K (32,768) words. Thus, memory addresses range from

Ø through $32{,}767_{1Ø}$
or Ø through $77{,}777_8$

and all memory cells can be addressed with a 15-bit word. The program counter, therefore, is 15 bits long; addresses stored in memory occupy bit positions 1 through 15.

Core Size (Words)	Memory Range	Address Range
4K (4,096)	Ø through $4{,}Ø95_{1Ø}$	Ø through 7777_8
8K (8,192)	Ø through $8{,}191_{1Ø}$	Ø through 17777_8
12K (12,288)	Ø through $12{,}287_{1Ø}$	Ø through 27777_8
16K (16,384)	Ø through $16{,}383_{1Ø}$	Ø through 37777_8
32K (32,768)	Ø through $32{,}767_{1Ø}$	Ø through 77777_8

The computer has four (4) 16-bit accumulators (ACØ, AC1, AC2, AC3) in which all arithmetic and logical functions are performed and through which all Input/Output transfers are made (except Data Channel transfers that bypass the active registers. See page 8-5.)

Associated with the accumulators is the Carry flag (carry bit, link, etc.) which indicates the occurrence of an arithmetic carry out of bit Ø.

The computer has one 15-bit Program Counter (PC) which contains the address of the memory location containing the next instruction. Following the execution of each instruction, the Program Counter is incremented by 1, resulting in a sequential program flow. Normal program flow may be altered by changing the contents of the Program Counter with a SKIP or JUMP instruction.

On the computer console are several switches, which allow manual entry of data into memory and into the accumulators and which control program and computer operation. There are also a number of indicator lights which display program, register, and accumulator information useful in program debugging.

The instruction set of the Nova family can be broken down in three classes:

Memory Reference Instruction Class (MRI): This class contains instructions which move data between the accumulators and memory, instructions which modify memory, and jump instructions which alter the program flow.

Arithmetic and Logical Instruction Class (ALC): This class contains instructions which manipulate the contents of accumulators and the Carry flag and instructions which perform all the arithmetic and logical functions between accumulators.

Input/Output Instruction Class (I/O): This class contains instructions which move data between the accumulators and the I/O peripheral devices and instructions which serve only to control the I/O devices.

CHAPTER 2

CENTRAL PROCESSING UNIT INSTRUCTIONS

MEMORY REFERENCE INSTRUCTIONS (MRI)

Each 16-bit MRI instruction word is divided into four (4) fields:

	bits
Command Field (C)	Ø through 4
Addressing Mode Field (I)	5
Index Field (X)	6 through 7
Displacement Field (D)	8 through 15

C	I	X	D
Ø 4	5	6 7	8 15

The command field determines the type of instruction: move data; modify memory; jump.

Every MRI must contain an effective address (E) which specifies which memory cell is to be referenced.

The effective address (E) is formed by the Index X and the Displacement D. The Index X refers to a register or accumulator to whose contents is added the displacement D, resulting in the address of the desired memory cell.

$$E = (X) + D \quad *$$

where ( ) means the "contents of."

If the Index (X) is Ø, then D is unsigned and may have the range ØØØ thru 377_8 (Ø thru 255_{10}).

With an index of 1, 2, or 3 specified, the displacement D is a signed number which takes on the values

positive:	ØØØ through 177_8	or	Ø through 127_{10}
negative:	377_8 through $2ØØ_8$	or	-1 through -128_{10}

In other words, if bit 8 (the left-most bit of D) is a Ø, D has a positive value. If bit 8 is a 1, D represents a negative displacement.

* Except when X = 1, see next page.

To obtain the effective address, D is added to the contents of the register or accumulator specified by the Index X.

If X is	Then (X) is	And The Effective Address E Has The Range
ØØ	Ø	Ø ≤ E ≤ 377_8. This is page Ø or base page addressing; it has fixed bounds and can be referenced from anywhere in memory.
Ø1	(PC)	(Present location -$2ØØ_8$) ≤ E ≤ (Present location +177_8). This is relative addressing; referencing is relative to the present location.
1Ø	(AC2)	[(AC2) - $2ØØ_8$] ≤ E ≤ [(AC2) + 177_8]. This is base register addressing and is solely a function of the (AC2) which is called a memory pointer.
11	(AC3)	[(AC3) -$2ØØ_8$] ≤ E ≤ [(AC3) +177_8].

If the Addressing Mode bit (I) is a Ø, then the addressing is direct, and the effective address points to the memory cell that is to be operated upon.

I = 1 specifies indirect addressing. That is, the effective address points to a memory cell whose contents form another effective address which replaces the old one.

In the memory word addressed, bits 1 through 15 are the new effective address, and bit Ø is the new I (Addressing Mode bit) which may be a Ø (the new E is a direct address) or a 1 (the new E is another level of indirect addressing).

<u>Example:</u>

If location 21Ø contains 1ØØ36Ø,
and location 36Ø contains ØØØ365
and location 365 contains 12345Ø

Then, if a fetch command has I = Ø, X = ØØ, D = 21Ø, the result is (21Ø) directly, which = 1ØØ36Ø.

But, if a fetch command has I = 1, X = ØØ, D = 21Ø, then:

(21Ø) becomes a new effective address with I = 1 and E = 36Ø,
so (36Ø) becomes another new effective address with I = Ø and E = 365,
and (365) is the result, which is 12345Ø.

This last example has two levels of indirect addressing.

Move Data MRI's

There are two MRI's which move data. One moves data from a memory cell into an accumulator; the other moves data from an accumulator into a memory cell. The MRI:

LDA	AC, D, X	loads accumulator AC with the contents of the memory cell specified by the effective address E, which is made up of the displacement D and Index X.
		AC can = Ø, 1, 2, or 3
		D can = -2ØØ $\leq$ D $\leq$ 177 (X = 1, 2, or 3) or = Ø $\leq$ D $\leq$ 377 (X = Ø or null)
		X can = null, Ø, 1, 2, or 3
		Null and 0 have the same effect.
STA	AC, D, X	stores the contents of accumulator AC into memory location E.

In an LDA instruction, the contents of location E are unchanged, and in an STA instruction, the contents of accumulator AC are unchanged.

Example:

Load AC3 from five (5) locations beyond where AC2 points.

LDA 3, 5, 2

now E = (AC2) +5.
If (AC2) = 1Ø6, then E=106+5=113 and (AC3) will = (113).

Example:

Store (AC3) in the $1ØØ_8$ location preceding our present location.

STA 3, -1ØØ, 1

now E = (PC)-1ØØ

Example:

To set the indirect bit to a 1 in order to cause indirect addressing, insert the @ symbol somewhere in the instruction statement:

If (AC2) = 1Ø6
and (113) = Ø1Ø4Ø7

then LDA 3,@5,2 loads AC3 with (1Ø4Ø7).

The same result could have been obtained with:

```
LDA   3,5,2   (AC3)=1Ø4Ø7
LDA   3,Ø,3   (AC3)=(1Ø4Ø7)
```

The last statement is valid because E is computed prior to the execution of the instruction.

Modify Memory MRI's

There are two MRI's which modify the contents of a memory cell. The MRI:

ISZ D,X — increments (E) by 1 and stores it back into E. If the new (E) = Ø, the next instruction in the program sequence is skipped. If the new (E) ≠ Ø, the normal program sequence is followed.

DSZ D,X — decrements (E) by 1 and skips if the new (E) = Ø.

Jump MRI's

There are two MRI's which alter the normal program sequence by jumping to an arbitrary location. One simply changes the (PC), and the other saves the old (PC) + 1 before changing the (PC). The MRI:

JMP D,X — loads E into the PC, takes the next instruction from location E, and continues sequential operation from there, i.e., transfers control to E.

JSR D,X — first computes E, then saves (PC) + 1 in AC3, then loads E into the PC, takes the next instruction from location E, and continues operation from there, i.e., transfers control to E and saves the former (PC)+1 in AC3.

Example:

Execute the routine LOOP ten (1Ø) times.

```
LOOP:   -------  }
        -------  }  executes this 1Ø₁Ø times
        -------  }
        DSZ COUNT
        JMP LOOP
        -------
        -------
COUNT:  12
```

Example:

Make a closed (self-initializing) subroutine called LOOP that is executed 10 times before control is returned to the main program. LOOP needs AC3 for computation.

```
              - - - - - - - -
              - - - - - - - -
              JSR LOOP
              (return here)
              - - - - - - - -
              - - - - - - - -
              - - - - - - - -

LOOP:         STA     3,SAVE             ;SAVE RETURN ADDRESS
              LDA     3,CONST            ;INITIALIZE COUNTER
              STA     3,COUNT
              - - - - - - - - - - - }
              - - - - - - - - - - - }    execute 1Ø₁Ø times
              - - - - - - - - - - - }
              DSZ     COUNT              ;SKIP IF EXECUTED 10 TIMES
              JMP     LOOP+3             ;EXECUTE AGAIN
              JMP     @SAVE              ;RETURN

SAVE:         Ø
CONST:        12
COUNT:        Ø
```

Example:

In this example, parameters DATA1 and DATA2 must be passed to LOOP, and return must be to the instruction following DATA2.

```
              - - - - - -
              JSR LOOP
              DATA1
              DATA2
              (return here)
              - - - - - -

LOOP:         STA     3,SAVE
              LDA     1,Ø,3              ;PUT DATA1 INTO AC1
              LDA     2,1,3              ;PUT DATA2 INTO AC2
              LDA     3,CONST
              STA     3,COUNT
              - - - - - - - - - - -
              - - - - - - - - - - -
```

(Example continued on next page)

```
                 DSZ     COUNT
                 JMP     LOOP+5
                 ISZ     SAVE
                 ISZ     SAVE
                 JMP     @SAVE
SAVE:            Ø
CONST:           12
COUNT:           Ø
```

Cannot say JMP @SAVE+2 because it would indirectly address via

There are a number of pre-defined memory cells:

Location Ø and location 1 are used by the hardware interrupt service routine. When an I/O device requests an interrupt, (PC) +1 are stored into location Ø and control is transferred by a JMP @1.

Locations 2Ø-27 are auto-incrementing locations. If addressed indirectly, their contents are first incremented by 1, then used as the effective address.

Locations 3Ø-37 are auto-decrementing locations.

Example:

Assume (2Ø) = ØØØ451.

The instruction STA 2,@2Ø,Ø will cause the contents of location 2Ø to be incremented to 452, and then (AC2) will be stored in location 452.

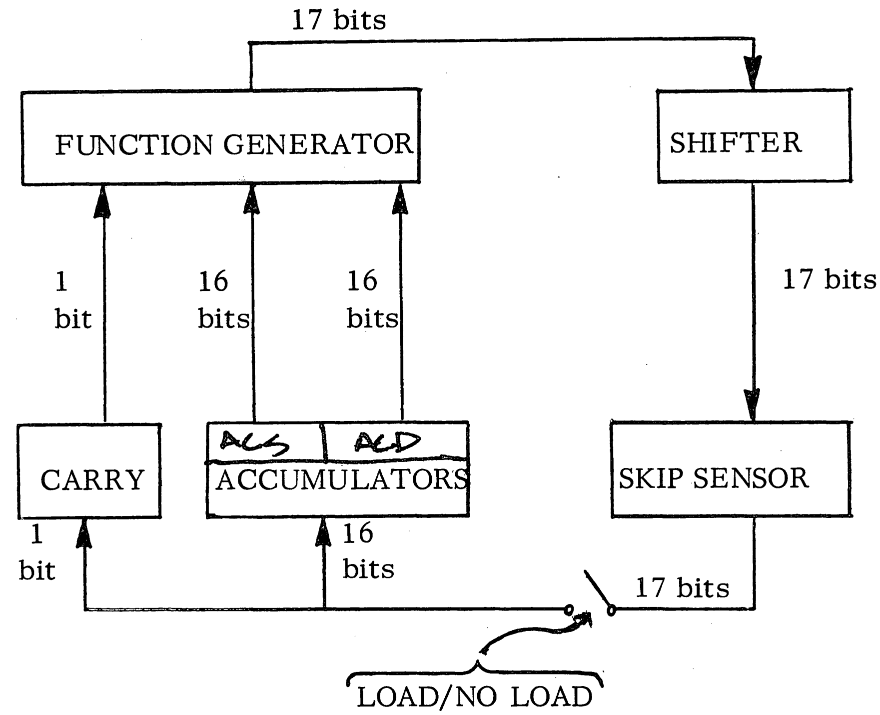
ARITHMETIC AND LOGICAL INSTRUCTIONS (ALC)

All arithmetic and logical processing is done between accumulators.

Example:

(ACØ) could be added to (AC1) with the sum left in ACØ. Also, (ACØ) could be added to (ACØ) with the sum left in ACØ.

This is possible because all processing is done external to the accumulators in the following manner.



Arithmetic and Logical Functions

The functional portion of all arithmetic and logical instructions is contained in bits Ø through 7 in the following manner:

1	SOURCE AC	DEST. AC	FUNCTION	
Ø	1 2	3 4	5 6 7	8 15

A 1 in bit Ø indicates an ALC.

The source AC is called ACS,and the destination AC is called ACD in text following.

The function is specified by 3 bits allowing $2^3 = 8$ possible functions:

COM	ACS, ACD	the 1's complement of ACS is deposited into ACD (1's complement = all 1's changed to Ø's and vice versa)
NEG	ACS, ACD	the 2's complement of ACS is deposited into ACD (then (ACD) = - (ACS))
MOV	ACS, ACD	copy (ACS) into ACD
INC	ACS, ACD	deposit (ACS) +1 into ACD
ADD	ACS, ACD	deposit (ACS) + (ACD) into ACD
SUB	ACS, ACD	deposit (ACD) - (ACS) into ACD
ADC	ACS, ACD	deposit (ACD) + 1's complement of (ACS) into ACD
AND	ACS, ACD	deposit (ACD) logical AND (ACS) into ACD

If ACS is not also ACD, then the original (ACS) are preserved.

Once the function has been performed, the result can be operated upon before it is loaded into ACD.

These additional operations make up bits 8 thru 15.

1	ACS	ACD	FNC	SHIFT	CARRY	NO LOAD	SKIP
Ø	1 2	3 4	5 6 7	8 9	1Ø 11	12	13 14 15

Carry Field

The value of the carry supplied to the function generator prior to performing the function is called the base value of the carry bit. This base value may be affected by the results of the function performed. If the function performed in the function generator results in an overflow, the base value of carry is complemented.

The following conditions will cause overflow:

Function	Unsigned Conditions Causing Overflow
ADD ACS, ACD	(ACS) + (ACD) $>2^{16}-1$
SUB ACS, ACD	(ACS) $\leq$ (ACD)
NEG ACS, ACD	(ACS) = Ø
INC ACS, ACD	(ACS) = $2^{16}-1$
ADC ACS, ACD	(ACS) $<$ (ACD)

The initial or base value of the carry bit is specified in the instruction by bits 1Ø and 11. (The mnemonic is appended to the 3 letter function mnemonic).

If the Function is Appended With	This Supplies
-	The current state of Carry to the function generator.
Z	A Ø to the function generator.
O	A 1 to the function generator.
C	The complement of the current state of Carry to the function generator.

Example:

SUB	Ø, Ø	clears ACØ; resultant Carry bit is unknown
SUBO	Ø, Ø	clears ACØ and Carry bit, since SUB causes an overflow which complements the C=1 base value.

Shift Field

After a function has been performed, its results may be rotated left or right as specified in the instruction by bits 8 and 9. (The mnemonic is appended to the 3 or 4 letter function and carry mnemonic, after the carry mnemonic, if one occurs.)

If the Shift Field is	The result is
-	unchanged
L	rotated left by 1 bit: bit 15 →14, 14 →13, ... , 1 →Ø, Ø →Carry, Carry →15
R	rotated right by 1 bit: bit 1 →2, 2 →3, ... , 14 →15, 15 →Carry, Carry →Ø
S	bytes are swapped: bits Ø thru 7 are swapped with bits 8 thru 15; Carry is unchanged.

Example: Perform a true shift left of (AC1) by one bit.

```
MOVZL    1,1
```

Skip Field

After the function has been performed and the results shifted, the results can be tested to see whether or not to conditionally skip the next instruction in sequence. The conditions are based on the specifications of bits 13, 14, and 15 in the instruction. (The following mnemonics can be used. They follow ACD with ,SKIP.)

If the mnemonic is	The shifted result will be tested and the program will
-	never skip
SKP	always skip
SZC	skip if the Carry bit is zero
SNC	skip if the Carry bit is non-zero
SZR	skip if the result (bits Ø thru 15) is zero
SNR	skip if the result (bits Ø thru 15) is non-zero
SEZ	skip if either the Carry or the result or both is zero.
SBN	skip if both the Carry and the result are non-zero

Load/No Load Field

Once the function has been performed, the shifting completed, and the decision for skip made, the result may or may not be loaded into ACD depending on bit 12 of the instruction.

LOAD/NO LOAD	BIT 12	RESULT IS
-	Ø	loaded into ACD
#	1	not loaded into ACD, leaving ACS and ACD unchanged.

Example:

Test for the sign of a number in AC1, but do not destroy it.

```
        MOVL#  1,1,SNC
        JMP    POS
        (negative)
        - - - - - - - - - - -
        - - - - - - - - - - -
        - - - - - - - - - - -
POS:    (positive)
        - - - - - - - - - - -
        - - - - - - - - - - -
        - - - - - - - - - - -
```

INPUT/OUTPUT INSTRUCTIONS (I/O)

The I/O instructions control all the operations between the processing unit and peripheral equipment. Every I/O instruction has the following format:

Ø11	AC	TRANSFER	CONTROL	DEVICE CODE
Ø 2	3 4	5 7	8 9	1Ø 15

Data Transfer

Any transferring of data is done between a particular device and a particular accumulator. The accumulator involved is specified by bits 3 and 4 (Ø, 1, 2, 3). The device involved is specified by the device code in bits 1Ø through 15. Bits 1Ø through 15 decode to 64 unique possibilities; however, only 62 devices may be addressed (Ø1 through 76_8). Device code ØØ is not used, and 77 is a special function code denoting the CPU. In a device, there may be up to 3 data buffers (A, B, and C). Bits 5 through 7, the transfer field, specify the buffer involved and the direction of the data transfer, whether IN or OUT. An IN transfer implies a data transfer from the device buffer to the processor. An OUT transfer implies a data transfer from the processor to the device buffer.

If the transfer field is	The transfer is	The mnemonic is
Ø	No I/O transfer	NIO
1	Data IN from buffer A	DIA
2	Data OUT to buffer A	DOA
3	Data IN from buffer B	DIB
4	Data OUT to buffer B	DOB
5	Data IN from buffer C	DIC
6	Data OUT to buffer C	DOC
7	(reserved for skip tests described later)	

Control Field

Once the device, buffer, and accumulator have been specified, it is necessary to send control information to the device via the control field, bits 8 and 9.

Associated with every device is a Busy and Done flip-flop. If both flip-flops are clear (reset), the device is in the idle mode. To place the device in operation, the Busy flip-flop must be set. After the device has processed the unit of data on a DATA OUT instruction, or when a device has information available in a buffer register on a DATA IN instruction, the Busy flip-flop is cleared and the Done flip-flop is set.

Using the control field in an I/O instruction, the following control functions can be specified by appending the appropriate mnemonic to the instruction.

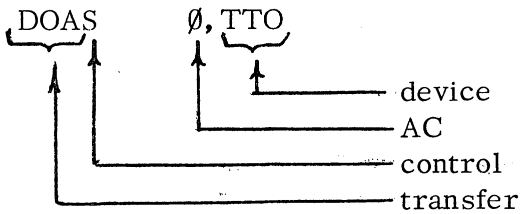
If the mnemonic is	The control function is
-	No control.
S	Set the Busy flip-flop and clear the Done flip-flop, thus starting the device.
C	Clear both the Busy and Done flip-flops, thus idling the device.
P	Special pulse output for customer applications; does not affect Busy nor Done flip-flop.

The general format of an I/O instruction is:

Transfer Control AC, Device Code

Example:

To type the character in ACØ on the teletype:



This instruction causes the contents of ACØ to be transferred to Buffer A of the TTO. The TTO is then started by the S in the control field, and the character is typed.

Example:

To idle the TTY output:

NIOC TTO

Example:

NIO TTO could be used as a no-op since there is no control or data transfer.

Special Functions

Using the special function transfer code 7, it is possible to test the status of the Busy and Done flip-flops and to conditionally skip the next instruction as a result of the test.

Mnemonic	XFR Code	Cont. Code	Function
SKPBN	7	Ø	Skip the next instruction if the Busy flip-flop is non-zero.
SKPBZ	7	1	Skip the next instruction if the Busy flip-flop is zero.
SKPDN	7	2	Skip if the Done flip-flop is non-zero.
SKPDZ	7	3	Skip if the Done flip-flip is zero.

Each skip-on-flag function must designate a specific device.

Example:

SKPDN	TTI	Tests the Done flag of the TTI.
SKPBZ	36	Tests the Busy flag of Device 36.

Example:

Read a character from the TTY, wait until it is in the Done state.

```
NIOS     TTI     ;START A READ CYCLE.
SKPDN    TTI     ;SKIP WHEN TTI DONE. (COULD BE SKPBZ TTI. )
JMP      .-1     ;CONTINUE SENSING STATUS.
DIAC     Ø,TTI   ;FETCH THE CHARACTER AND IDLE TTI.
```

The following two special functions, that use a device code of 77, are of particular interest:

Mnemonic	Description
READS AC ≡ DIA AC, CPU	Reads the contents of the console data switches and deposit in AC.
HALT ≡ DOC Ø, CPU	Halt the processor.

CHAPTER 3

BASIC ASSEMBLER

The assembler allows the programmer to write his program in a symbolic mnemonic language as opposed to direct octal numeric coding. The instruction format for the assembler separates a line into four possible fields:

LABEL: OPCODE OPERAND ;COMMENT

LABEL

Labels must be of the form abbbb: where:

a = A-Z and .
and b = A-Z and Ø-9 and .

Example:

```
DONE:
.DABS:
.EC9:
```

A label may contain one or more characters, but only the first five characters are retained by the assembler, and labels whose first five characters are the same are considered identical. Note: The period (.) cannot be used alone as a label.

Example:

SQUARE ≡ SQUARE.ROOT ≡ SQUAR
SQU ≢ SQUAR

The label may occur in any column, and all labels must terminate with a colon.

OPCODE

The opcode may follow immediately after the colon of a label, or in the event that no label precedes it, the opcode may begin in any column.

OPERAND

The operand must be separated from the opcode by atleast one space, one comma, or one TAB. There maybe up to 3 operands, but each must be separated from the other by at least one space, one comma, or one TAB.

Example:

LDA   Ø,1,2   ≡   LDA   Ø,,,,,1 2

Example:

LDA   3,Ø,2   ≢   LDA 3,,2   but   LDA 3,2   ≡   LDA 3,,2

COMMENT

The comment field must always be preceded by a semicolon. If a comment is to be carried to a new line, a semicolon must be the first character of the comment on the new line. Any character may appear in a comment field except a CR or an FF (Carriage Return or Form Feed).

PROGRAM FORMAT

Every statement must end with a CR. Do not forget the CR after the .END statement, which must be the last statement of the symbolic program.

The assembler contains pre-defined settings at every eighth space from the beginning of a line: columns 1, 9, 17, 25, ...

When the TAB key is depressed (CTRL I) or when the TAB code appears on the paper tape, the software spaces to the next tabulation column.

The assembler automatically segments the listing into pages that are 11 inches long. The top of each page is indicated on the listing by the appearance of three underscore characters in the upper left-hand corner. It is possible for the programmer to force the assembler to begin a new page in the listing by merely inserting a form feed character in the input source tape. If the new page was begun as a result of a program-executed FF, the three underscores will appear as follows:

$$\overline{\uparrow}\ \overline{\uparrow}\ \overline{\uparrow}$$

In arithmetic expressions, there is no hierachy of operations; they are performed from left to right, with no parentheses allowed. Allowable operators are:

+ - * / & !   where: & = ∧ = AND   ! = ∨ = OR

SPECIAL CHARACTERS

There are four special characters: .   "   @   #

. indicates the current location or contents of the program counter.

Example:

JMP .+1 ≡ JMP (PC)+1 or jump to the next instruction.

Example:

C=. assigns the present contents of PC to the symbol C.

" replaces the next character by its ASCII equivalence. (Does not apply to RUBOUT, LINE FEED, FORM FEED, or NULL.)

Example:

A: "A stores the ASCII equivalence of the character A into the cell labeled A.

Example:

Make up a memory cell

WORD: | D | C |

could be done by: WORD: "D*4ØØ+"C

@ places a 1 in the indirect bit of an MRI or an address word. This 1 bit is OR'ed (V) with the assembled instruction after the rest of the instruction has been assembled.

Example:

LDA@ Ø,Ø,1 is assembled as

Ø2Ø4ØØ V ØØ2ØØØ = Ø224ØØ

ØØ2ØØØ is the value of @ for an MRI instruction.

Example:

If AGET: @JOE and JOE=Ø27351

then AGET will contain Ø27351 V 1ØØØØØ = 127351 because 1ØØØØØ is the value of @ for an address word.

places a one in the no load bit of an ALC. As with @, the 1-bit is OR'ed with the assembled instruction.

Example:

ADDL 1,2,SZC has Ø in bit 12

1	Ø1	1Ø	11Ø	Ø1	ØØ	Ø	Ø1Ø

ADDL# 1,2,SZC has 1 in bit 12

1	Ø1	1Ø	11Ø	Ø1	ØØ	1	Ø1Ø

The first instruction assembles as 1331Ø2 and places the sign of the sum in Carry, the rest of the sum in bits Ø-14 of AC2, and either Ø or 1 in bit 15 of AC2, depending upon whether or not there is a carry out of the sign bit. The instruction causes the processor to skip the next instruction on a positive sum.

The second instruction assembles as 133112 and causes the processor to skip the next instruction on a positive sum without affecting either Carry or AC2.

ASSEMBLER PSEUDO-OPS

A number of pseudo-op instructions are associated with the assembler. These symbols relay commands to the assembler.

.LOC Expression

This pseudo-op is used to set the contents of the location counter to the value determined by the expression. If this pseudo-op does not appear in the symbolic program, the program will be assembled starting at location Ø.

Example:

If it is desired that the program be assembled starting at location 400, the first statement of the program should be:

```
.LOC    4ØØ
```

The contents of the location counter may be changed at any point in the program with a .LOC statement.

Example:

Reserve a block of ten locations for a table whose first location is TBL1.

```
TBL1:    .LOC    .+12
```

.BLK Expression

This pseudo-op is used specifically to reserve a block of storage. It is important to note that the block of storage reserved is not initialized to zero.

Example:

The previous example may be written in the following manner:

```
   TBL1:    .BLK    12
or TBL1:    .BLK    2*5
```

.RDX Expression

At the beginning of each pass, the assembler is initialized to interpret all integers as octal. The radix can be changed at any time by using the pseudo-op:

```
.RDX    Expression
```

The expression must evaluate to an integer between 2 and 1Ø and any integers in the expression are always interpreted as being decimal. Once a change of radix has been made, all succeeding integers are interpreted to that radix until such time as another radix change is made or the present assembler pass is completed.

Example:

To reserve a block of ten locations:

```
        .RDX    1Ø
TBL1:   .BLK    1Ø
        .RDX    8
        - - - - - - -
        - - - - - - -
        - - - - - - -
```

. TXT *message*

Character strings may be stored, 2 characters to a word, by using the . TXT pseudo-op followed by the character string enclosed by text delimiter characters, which are any characters other than those contained within the string, CR, TAB, comma, space, null, LF, FF or RUBOUT. Within the text message the assembler ignores CR, FF, LF, RUBOUT, and NULL characters.

Examples:

. TXT   *ABCDEFG*   gets stored as

Ø	8
B	A
D	C
F	E
null	G

. TXT   *ABCD*   gets stored as

B	A
D	C
null	null

Normal packing is from right to left; however, the packing may be changed to left to right at any time by a . TXTM pseudo-op described below. The packing remains in the new mode until altered again or assembly terminates.

Prior to packing, each character is reduced to only the low order 7 bits of the ASCII code. The 8th (leftmost) bit can be selected by using the following text pseudo-ops:

. TXT	left bit is always Ø
. TXTF	left bit is always 1
. TXTO	left bit is odd parity
. TXTE	left bit is even parity

Within the text string, any character can be introduced by enclosing it within angle brackets.

Example:

. TXT   *<33>*   puts in just the code 33

. TXTM Expression

If the expression evaluates to zero, packing is from right to left. If the expression evaulates to non-zero, packing is from left to right.

Source Program Termination Pseudo-ops

Every source program tape must end with one of the following pseudo-ops:

.END	This causes the assembler to put a START block at the end of the object tape, forcing the Binary Loader to halt when the object tape has been loaded.
.END expression	This causes the assembler to put a START block at the end of the object tape, forcing the Binary Loader to transfer control to the location specified by Expression after the object tape has been loaded.
.EOT	This tells the assembler that there is another tape for this source program. After encountering this pseudo-op, the assembler will halt, allowing the operator to load the next tape. When the CONTINUE switch is depressed, the assembly will continue.

Symbol Table Pseudo-ops

By using symbol table pseudo-ops, the user can define new instruction mnemonics and assign names to constants.

.DUSR

This pseudo-op is used to define symbols which, when used in the source program, require no additional arguments.

Example:

Define the following constants:

```
.DUSR          TEN=12
```

now can be used as:

```
LDA            1,TEN,3
```

and is equivalent to:

```
LDA            1,12,3
```

Example:

Define a no-op symbol.

```
.DUSR       NOOP=MOV    Ø,Ø
            - - - - - - - -
            - - - - - - - -
            - - - - - - - -
            NOOP
            - - - - - - - -
            - - - - - - - -
            - - - - - - - -
```

Example:

Define a symbol that will type the character in ACØ

```
.DUSR       TYPEØ=DOAS   Ø,TTO
```

Use the symbol in place of the instruction:

```
SKPBZ       TTO
JMP         .-1
TYPEØ
```

Define a symbol that will type the character in AC1:

```
.DUSR       TYPE1=DOAS   1,TTO
```

.DMR

This pseudo-op is used to define MRI's which do not require an accumulator as an operand. An example of such an instruction is JMP.

Example:

Define a symbol GOTO that acts exactly like the JMP instruction.

```
.DMR        GOTO=JMP     Ø
```

can be used as:

```
GOTO    ABC
```

and is equivalent to:

```
JMP     ABC
```

.DMRA

This defines MRI's which require an accumulator as an operand. An example of an instruction of this type is LDA.

Example:

Define a symbol LOAD that acts exactly like the LDA instruction.

```
.DMRA        LOAD=LDA    Ø,Ø
```

can be used as:

```
LOAD         3,XYZ
```

and is equivalent to:

```
LDA          3,XYZ
```

.DALC

This defines arithmetic and logic class symbols.

Example:

Define a symbol which will skip the next instruction if (ACS) >(ACD).

```
.DALC        SGT=SUBL#     Ø,Ø,SNC
```

can be used as:

```
SGT       2,3
```

Skips if (AC2) >(AC3) and saves the contents of both.

For Unsigned Numbers:

```
SUBZ#        1,Ø,SZC     ;SKIP IF ACØ  < AC1
ADCZ#        1,Ø,SZC     ;ACØ ≤ AC1
```

For Signed Numbers:

```
SUBL#        ACS,ACD,SNC    ;ACS > ACD
ADCL#        ACS, ACD,SNC   ;ACS ≥ACD
SUBL#        ACS,ACD,SZC    ;ACS <ACD
ADCL#        ACS,ACD,SZC    ;ACS ≤ACD
```

.DIO

This defines I/O symbols in which only a device code is required as an operand.

Example:

Define a symbol which will issue a start pulse to a device.

```
.DIO        STT=NIOS    Ø
```

.DIOA

This defines I/O symbols that require both an accumulator and a device code as operands.

Example:

The DOA instruction is defined as shown.

```
.DIOA       DOA=Ø61ØØØ
```

.DIAC

This defines an instruction that requires one operand which will replace bits 3 and 4 with an accumulator number.

Example:

The INTA instruction is defined as shown:

```
.DIAC       INTA=DIB    Ø,CPU
```

.DALC symbols can be appended with any one of the following 4th characters:

L R S Z O C (Carry and shift fields. See pages 2-8 and 2-10).

and any one of the following 5th characters:

L R S (Shift field after carry field. See pages 2-9 and 2-10).

.DIO and .DIOA can be appended with:

S   C   or P   (Control field. See page 2-12.)

Hence, care must be taken in selecting the fourth and fifth letters of .DALC, .DIO, and .DIOA symbols. See Assembler Manual (093-000017) for details.

The user defined symbols become part of the initial symbol table which also contains the instruction mnemonics and the permanent symbols. The initial symbol table can be reduced to its initial size only by reloading the assembler.

By using the pseudo-op .XPNG, all user defined symbols and instruction mnemonics are erased from the initial symbol table leaving only the permanent symbols such as . and the pseudo-ops. This enables the programmer to redefine even the instruction mnemonics. But since there are no symbols in the table, new initial symbols must be defined numerically.

The following ASCII characters are ignored by the assembler:

NULL	-	ØØØ
RUBOUT	-	377
LF	-	Ø12

ASSEMBLER OPERATING PROCEDURES

When the assembler is loaded, it requests information on the input and output devices to be used and on the assembly mode. The assembler can be restarted at location ØØØØ2, and new I/O assignments can be made. Restarting at location ØØØØ3 initiates only a new MODE request. The assembler queries are given below.

IN:

The user responds to the input device request with a single digit naming the device. The following numbers may be given:

1 Teletype reader without parity checking
2 Teletype reader with parity checking
3 Paper tape reader without parity checking
4 Paper tape reader with parity checking
5 Teletype keyboard without parity checking

LIST:

The user responds to the listing device request with a single digit naming the device. The following may be given:

1 Teletype Model 33
2 Teletype Model 35
3 Line printer
4 Paper tape punch (for ASR33)
5 Paper tape punch (for ASR35)

BIN:

The user responds to the binary output device request with a single digit naming the device. The following may be given:

1 Teletype punch without local symbols
2 Paper tape punch without local symbols
3* Teletype punch with local symbols
4* Paper tape punch with local symbols

MODE:

The user responds to the assembly mode request with a single digit naming the mode. The following may be given:

1 Pass 1
2 Pass 2 - Output object tape
3 Pass 2 - Output listing
4 Pass 2 - Output object tape and listing
5 Output symbol list

* Only when using the Relocatable Assembler. See Chapter 9.

CHAPTER 4

TEXT EDITOR

The editor enables text editing on previously prepared files as well as generating and editing new files. Once loaded, the editor is self-starting and, in a 4K machine, provides over 3000 characters of text storage. This is approximately three 8-1/2 x 11 pages of normal symbolic source program text.

The editor logically segments the input string of characters into smaller subdivisions for ease of editing. The entire input file is first segmented into pages where a page is defined as a string of characters up to, <u>but not including</u>, an FF character. Pages are further segmented into lines where a line is defined as a string of characters up to, and <u>including</u>, a carriage return.

Once a file has been loaded into the text buffer, an implicit character pointer (CP) is located within the text. It is the location of this CP that references all editing processes. The CP can be considered as always pointing between two characters in the edit buffer.

The normal editing procedure is to input a page, edit the text, and output the edited page.

When loaded, the editor first requests I/O device information, i.e.,

TTO (1) OR PTP (2) ?
TTI (1) OR PTR (2) ?
OUTPUT PARITY (1) OR NOT (2) ?
INPUT PARITY (1) OR NOT (2) ?

All editing commands are given from the teletype keyboard.

The following I/O commands are available:

Y	Yank a page into the edit buffer from the input device. Old contents of edit buffer are erased.
A	Yank a page into the edit buffer from the input device, and append it to the present contents of the edit buffer.
T	Type the entire edit buffer on the TTY.
<u>n</u>T	Type <u>n</u> lines of the edit buffer on the TTY starting at the current position of CP.
P	Punch the entire edit buffer on the output device, and follow it with a FF.

nP Punch n lines of the edit buffer on the output device, and follow it with a FF. Punches from current position of CP.

PW Punch the entire edit buffer on the output device, and do not follow it with a FF.

nPW Punch n lines of the edit buffer on the output device, and do not follow it with a FF. Punches from current position of CP.

nR Perform a P followed by a Y (PY) n times.

F Punch a FF on the output device.

nF Punch n inches of leader tape (null code) on the output device.

The following editing commands are available:

B Position the CP to the beginning of the edit buffer.

Z Position the CP to the end of the edit buffer.

nJ Position the CP n lines from the beginning of the edit buffer.

nL Position the CP n lines from the current position of the CP.

nM Position the CP n characters from the current position of the CP.

nK Delete (kill) n lines from the current position of the CP.

nD Delete n characters from the current position of the CP.

XM$command_1 \ldots command_n$\$ Define the macro-command string.

nX Execute the previously defined macro-command string n times.

XD Delete the previously defined macro-command.

Sstring$ Search for the string beginning at the current position of the CP, and reposition the CP after the last character of the "found" string.

Cstring1 \$string$_2$ \$ Search for string 1 beginning at the current position of the CP; when found, replace string 1 by string 2, and reposition the CP after the last character of the "inserted" string.

Istring$	Insert the string starting at the CP, and reposition the CP after the last character of the "inserted" string.

In addition, the following special commands are available:

=	Print the number of characters in the edit buffer on the TTY.
:	Print the number of lines in the edit buffer on the TTY.
.	Print the line number on the TTY where CP is now positioned.
RUBOUT	Erase the last character (will echo character deleted)
CTRL C	Return control to the Editor.

The editor ignores from text input:

NULL	ØØØ
RUBOUT	377 (not ignored in a command string)
LF	Ø12

Restarting the editor at location ØØØØ2 initiates a request for new I/O assignments. Restarting at location ØØØØ3 initiates only a command request.

The command delimiter is the ESC (escape) key, which is echoed as a $. Any number of commands may be included within a command statement, with no additional delimiting required, over and above that specified for the individual commands. Execution of a command string is not performed until the string is terminated with two (2) consecutive ESC characters. All numeric arguments are in decimal.

Example:

Search the edit buffer for a string, insert a statement before this string, print the new statement and the one following it. Then search the buffer for another string, change it, print the line containing the new string, and punch the contents of the buffer with a leader and trailer tape.

```
Buffer start
 Search on GETC:
              Reset CP to beginning of line
               Insert ";INPUT ROUTINE↵"

BSGETC:$ØLI;INPUT ROUTINE↵
$-1L2TBCPUT:$PUTC:$ØL1T25FP25F$$

                             Punch 25 inches of trailer
                           Punch entire buffer
                          Punch 25 inches leader
                       Type one line
                     Reset CP to beginning of line
      Change PUT: to PUTC:
     Buffer start
    Type two lines
  Reset CP to beginning of previous line
```

When initially loaded, the editor will simulate a software TAB. In other words, the formatting switch within the editor is "set". The condition of this switch can be complemented at any time by typing CTRL P.

After loading the editor, tapes may be created at the keyboard in the following manner:

```
I
- - - - - - -
- - - - - - -
- - - - - - -
$25FP25FB2ØØØK$$
I
- - - - - - -
- - - - - - -
- - - - - - -
```

CHAPTER 5

CONSOLE OPERATION

POWER SWITCH

The power switch has three positions:

OFF - Power off position.
ON - Power on, normal operation mode.
LOCK - Power on, operating switches disabled, only data switches enabled.

INDICATORS

The Nova family of computers have the following indicators on the console:

INSTRUCTION	-	Display left 8 bits (Ø through 7) of most recently executed instruction. (Nova and Supernova only).
ADDRESS	-	Display contents of PC.
DATA	-	Display data written in last MRI. Supernova: Following a memory step, the data indicators display the address for the next reference.
RUN	-	Processor is running.
ION	-	Program interrupts are enabled.
FETCH	-	Next cycle will fetch an instruction from memory.
DEFER	-	When executing an indirect addressing instruction, next cycle will fetch an address word from memory.
EXECUTE	-	When executing a move data or modify memory instruction, next cycle will fetch operand from memory.

The following additional indicators are on the Nova and Supernova:

DCH	-	Next cycle will be used by data channel for I/O.
PI	-	Next cycle will start a program interrupt.

The following additional indicators are on the Supernova:

OVERLAP	-	When executing ALC class instructions from read-only memory, execute/fetch overlapping is occurring.

PROTECT - (Used with memory protect option).

SWITCHES

When in RUN mode, only data switches and the STOP/RESET switch are enabled. The switches have the following functions:

ACØ DEPOSIT	-	Loads the contents of the data switches in ACØ.
ACØ EXAMINE	-	Displays the contents of ACØ in the data lights.

(ACl, AC2, and AC3 operate like ACØ in deposit and examine positions.)

EXAMINE	-	Loads the address contained in the data switches into PC and displays the contents of that address location in the data lights. (PC is displayed in the address lights).
DEPOSIT	-	Deposits, and displays in the data lights, the contents of the data switches into the address location specified by the address lights.
EXAMINE NEXT	-	Increments PC by 1, then performs an EXAMINE.
DEPOSIT NEXT	-	Increments PC by 1, then performs a DEPOSIT.
STOP	-	Finishes the current instruction, then stops the processor by turning off the clock. (Resets the RUN flip-flop.)
RESET	-	Performs a STOP, resets all I/O flags and disables interrupt. (Resets the ION flip-flop).
START	-	Loads the contents of the data switches into PC and begins normal operation by executing the instruction at the location specified by PC. (Sets the RUN flip-flop).
CONTINUE	-	Begin normal operation in the state indicated by the lights.
INST. STEP	-	CONTINUE for one instruction cycle.
MEMORY STEP	-	CONTINUE for one processor cycle. (An instruction cycle can be comprised of up to five (5) processor cycles).

There is an additional switch on the Supernova, Nova 800, and Nova 1200:

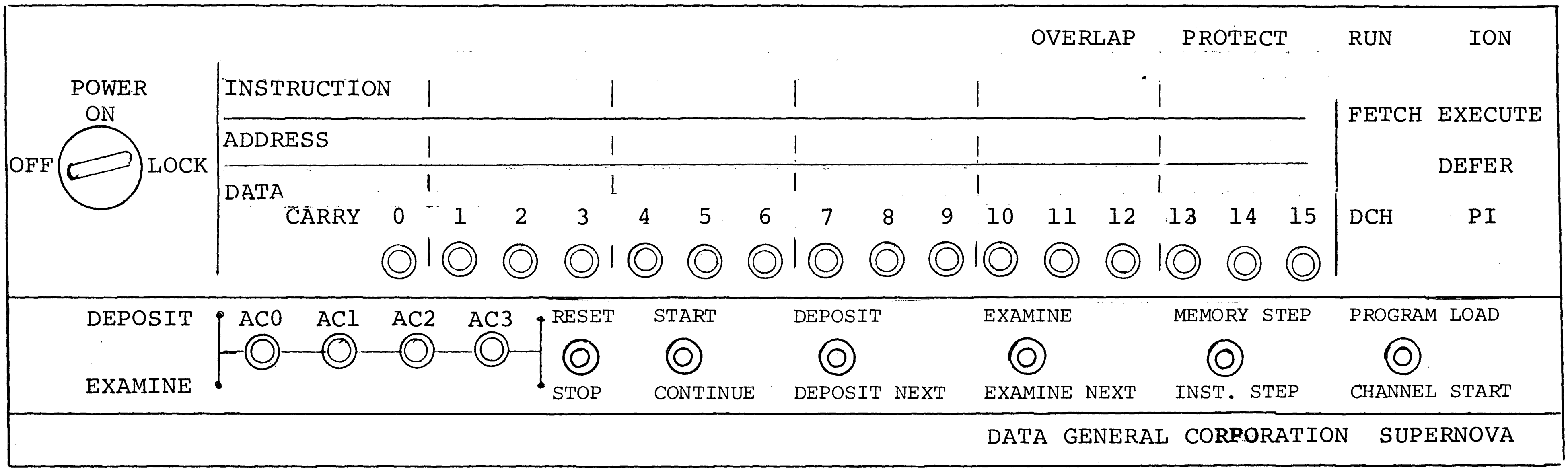
PROGRAM LOAD	-	This is a hardware loader that loads the Binary Loader from a special tape.

There is an additional switch on the Supernova:

CHANNEL START - Can be used to start a data channel input sequence manually.

It is good practice to RESET before START.

The figure on the page following shows the operator console of a Supernova that has all indicators and switches. The consoles for other Nova-line computers are similar.



SUPERNOVA OPERATING CONSOLE

CHAPTER 6

PROGRAM LOADERS

BOOTSTRAP LOADER

The Bootstrap Loader is a very short program which is manually loaded into memory. When executed, this program reads in a more sophisticated loading program called the Binary Loader. It is the Binary Loader which is used to load all other program tapes into memory.

The Bootstrap Loader eliminates the need for manually entering a much longer loading program.

The Bootstrap Loader program for Paper Tape Reader input is shown below. To change this version for Teletype input, change the underlined portions of the program as follows:

Change PTR to TTI

Change 12 to 1Ø

```
Ø7757   12644Ø GET:     SUBO 1,1          ;CLEAR AC1,CRY
Ø776Ø   Ø63612          SKPDN PTR
Ø7761   ØØØ777          JMP .-1           ;WAIT FOR DONE FLAG
Ø7762   Ø6Ø512          DIAS Ø,PTR        ;READ INTO ACØ AND RESTART READ
Ø7763   1271ØØ          ADDL 1,1          ;SHIFT AC1 LEFT
Ø7764   1271ØØ          ADDL 1,1          ;4 PLACES
Ø7765   1Ø7ØØ3          ADD Ø,1,SNC       ; ADD IN THE NEW WORD
Ø7766   ØØØ772          JMP GET+1         ;FULL WORD NOT ASSEMBLED YET
Ø7767   ØØ14ØØ          JMP Ø,3           ;OK, EXIT

                ;BOOTSTRAP LOADER STARTS HERE
Ø777Ø   Ø6Ø112 BSTRP:   NIOS PTR          ;START THE READER
Ø7771   ØØ4766          JSR GET           ;GET A WORD
Ø7772   Ø444Ø2          STA 1,.+2         'STORE IT TO EXECUTE IT
Ø7773   ØØ4764          JSR GET           ;GET ANOTHER WORD
                                          ;THIS WILL CONTAIN A STA INSTR
                                          ;THIS WILL CONTAIN JMP .-4

                .END
```

In order to reduce the possibility of erasing the Bootstrap Loader, it is common practice to load it into upper core. Thus, the Bootstrap Loader is entered beginning at location X7757 where X is the number of 4K memory blocks available over and above the initial 4K block.

Example:

Machine Core Size	Starting Address of The Bootstrap Loader
4K	Ø7757
8K	17757
12K	27757
16K	37757
20K	47757
24K	57757
28K	67757
32K	77757

Once the Bootstrap Loader is in memory, the operator loads the Binary Loader tape, 091-000004, into the reader, turns the reader ON, sets the data switches to x777Ø, presses RESET then START. The Binary Loader program will then be read into memory.

On the SUPERNOVA, there is no need to manually load the Bootstrap Loader. It is necessary only to load the special Self-Load Bootstrap and Binary Loader tape, 091-000041, into the reader, set the data switches on the console to contain the device code of the reader, turn the reader ON, and press PROGRAM LOAD. The Binary Loader program will then be read into memory.

The NOVA 800 and NOVA 1200 have an optional self-load feature. The Bootstrap Loader is part of the NOVA hardware and the Binary Loader is on a special self-load tape. It is necessary only to load the special Binary Loader tape, 091-000036, into the reader, set the data switches on the console to contain the device code of the reader, turn the reader ON, and press PROGRAM LOAD. The LSI chips containing the Bootstrap Loader are deposited in memory and the Binary Loader tape is then read into memory.

BINARY LOADER

The Binary Loader program loads all absolute object tapes into memory and resides in core locations x7646 through x7777. It is common practice to write programs which do not alter these locations, thus eliminating the need to reload the loaders. In all but very rate instances, DGC standard software is written so as not to destroy the Binary or Bootstrap Loader programs. In no case will any of this software destroy the Bootstrap Loader program.

To load object format tapes using the Binary Loader, load the object tape into the reader, turn the reader ON, set the switches to x7777, set data switch Ø to specify the reader in use (down for TTY reader, up for PTR), press RESET and START.

If the End Block on the object tape specifies a starting address of the program, the Binary Loader will transfer control to that location once the tape is loaded. Otherwise, load the starting address of the program into the data switches, press RESET then START.

CHAPTER 7

NUMERIC DEBUGGER

There are two numeric debuggers and a symbolic debugger that can be used in debugging programs on NOVA-line computers. The debuggers are called Debug I, Debug II, and Debug III. Debug II is described in detail here.

DEBUG I

Debug I is a stripped down version of Debug II. Debug I requires so little resident memory (about $2ØØ_{1Ø}$ locations) that in most cases it can be left resident in core.

Debug I provides for one breakpoint; examination and modification of the accumulators, Carry and memory from the TTY; and monitoring of the machine state.

DEBUG III

Debug III is a fully symbolic debugger which allows for the addressing of a program during the debugging process using the same symbols that were defined at assembly time. Debug III provides for up to eight breakpoints.

DEBUG II

Although Debug II requires about $8ØØ_{1Ø}$ resident core locations, it has an abundance of additional features over and above those available with Debug I.

Debug II provides for up to four breakpoints; examination and modification of the accumulators, Carry, and memory from the TTY; monitoring of the machine state; expression evaluation; punching of memory in binary format; and memory searches.

Examining and Modifying Registers

To examine an accumulator, type:

> nA

where n is the specified accumulator. The debugger response is /DDDDDD where the D's represent the contents of the accumulator. If no modification is desired, type a CR. To modify (AC), simply type in the new contents, then a CR.

The new contents may be expressed as an octal number or an octal expression of the form:

> $\pm$ octal no. $\pm$ octal no. $\pm$ - - - - -

In addition to octal numbers in an expression, a $ may be used. The $ will be replaced with the current contents of the examine register.

Similarly, Carry can be examined and modified by typing:

C

Any memory location can be examined and modified by typing:

address/

Typing ./ opens the register most recently closed. If a memory location examination or modification is followed by a CR, that register is closed. Replacing the CR with a LF (line feed) closes the register and opens the succeeding one. Replacing the CR with an ↑ closes the register and opens the preceding one.

To modify a memory location without first examining it, type

address!

<u>Searching Memory</u>

All memory or portions of memory may be searched for those locations with certain contents. The portions of memory to be searched are given by:

starting address, ending address S

When the search finds a core location that contains the information being searched for, the memory location and its contents will be printed. The contents of the M(mask) and W(word) registers determine the information to be searched for. These two registers may be examined and modified in the same manner as an accumulator.

The search is conducted by taking the contents of the current memory location and ANDing that word with the contents of the M register. If the result is equal to the contents of the W register, a match is said to occur, and the memory location and its contents are printed.

<u>Example:</u>

Find all occurrences of the word 132675 in the range of memory from location 4Ø through location 757.

Examine the M register, and modify it to contain all 1's, i.e. 177777.

Examine the W register, and modify it to contain the word being searched for: 132675.

The command string 4Ø, 757S will cause locations 4Ø through 757 inclusive to be searched for the 132675. Each time the equation:

$$W = (\text{Location being searched}) \wedge M$$

is satisfied, the location and its contents will be printed.

<u>Example:</u>

A routine resides in locations 4ØØ through 563. Find all DOA 1Ø instructions so that they can be changed to DOA 11.

Since DOA 1Ø = Ø 11x xØ1 Øxx ØØ1 ØØØ

examine and modify M to be:

1 11Ø Ø11 1ØØ 111 111 = 163477

Examine and modify W to be:

Ø 11Ø ØØ1 ØØØ ØØ1 ØØØ = Ø61Ø1Ø

The debugger command 4ØØ, 563S can now be given to search memory.

If a complete octal listing of a range of memory is desired, set M and W equal to Ø. Because (address)∧Ø=Ø, every location produces a match and is printed.

<u>Breakpoints</u>

In the debugging of routines, it is very helpful to be able to execute small portions of the routine and then examine various registers. Four breakpoints are provided for this purpose. A breakpoint is essentially a HALT command given at a certain point in a routine.

To set a breakpoint, type:

<u>address</u> B

The debugger will then replace the contents of the address with:

JMP @1<u>n</u>

when the routine is executed. The debugger replaces <u>n</u> with the number of the breakpoint (Ø through 3), and locations 1Ø through 13 contain debugger re-entry points.

When a breakpoint is encountered, the original contents of the location are replaced, and the following response is made:

```
address Bn
(ACØ)      (AC1)      (AC2)      (AC3)
```

where n is the number (Ø-3) of the breakpoint encountered.

All breakpoints may be examined by   B.

All the breakpoints may be deactivated by   D.

Any one breakpoint may be deactivated by   nD.

Control may be transferred to any address by typing:

```
addressR
```

If the address is not specified, the debugger goes to register L and uses (L) as the address. L may be examined and modified as is an accumulator.

If the routine halts at a breakpoint, control can be returned to the routine after the breakpoint by typing:

```
P
```

If the breakpoint is within a loop, a routine break can be set at the nth occurrence of that breakpoint by typing:

```
nP
```

Punching an Object Tape

Once a program has been debugged, a new binary object tape can be punched, with leader and trailer tape, by issuing the following commands:

```
nF
address1,address2P
E
nF
```

If an auto-start block is desired, replace the command E with addrE, where addr is the desired starting address.

Calculations using Debug II

In addition to the debugging capabilities, Debug II has a special command of the form:

<u>expression=</u>

The expression may be an addition and/or subtraction of octal numbers, a ., or a $.

<u>Example:</u>

561-472+3-5 = returns 65
. = returns the address of the last closed memory register.
$ = returns the contents of the last closed register (whether memory, accumulator, breakpoint, etc.)

This feature is handy as an octal arithmetic scratchpad calculator.

CHAPTER 8

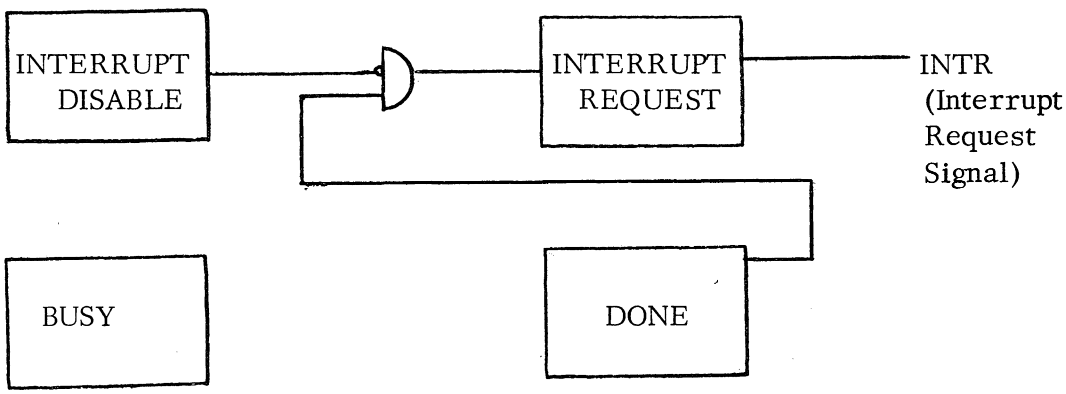
I/O DEVICE HANDLING

PROGRAM INTERRUPTS

Although peripheral devices may be serviced by the processor on a dedicated basis, as previously discussed, this usually results in extremely inefficient use of processor time and/or temporary neglect of all other devices.

To overcome this, a device interrupt and servicing facility is available. This facility provides for enabling and disabling devices from requesting service, establishing 16 levels of priority interrupts, and servicing devices only when they request service.

In addition to the BUSY and DONE flip-flops, every device has an Interrupt Disable flip-flop and an Interrupt Request flip-flop arranged logically as follows:



Within the processor is an interrupt system status flag (ION). When the flag is reset, indicating that the interrupt system is disabled, no device can interrupt the processor. When the flag is set and the interrupt system is on, selected devices may request service via an interrupt.

The interrupt system is enabled by the instruction INTEN (NIOS CPU) and disabled by the instruction INTDS (NIOC CPU). The status of the interrupt system can be monitored by the ION indicator on the front panel or by the instructions:

SKPBZ CPU	SKIP NEXT INSTRUCTION if interrupts are disabled.
SKPBN CPU	SKIP NEXT INSTRUCTION if interrupts are enabled.

The CPU hardware prevents all devices from interrupting when the ION flag is reset. In addition, a particular device cannot request an interrupt if its Interrupt Disable flip-flop is set.

Thus, the following conditions must be met before a device can interrupt the processor.

1. The ION flag must be set. (Interrupts enabled).

2. The device's Interrupt Disable flip-flop must be reset. (Interrupts allowed from the device.)

3. The device's DONE flip-flop must be set. (Device is ready for service.)

The commands for controlling the ION flag are:

INTEN	Interrupt Enable (set ION flag)
INTDS	Interrupt Disable (reset ION flag)

The command for controlling the individual Interrupt Disable flip-flops is:

```
MSKO   AC     ;MASK OUT
```

When an MSKO AC command is given, the Interrupt Disable flip-flop of every device is effectively connected to one of the 16 bit positions in accumulator AC. If the bit position contains a 1, all Interrupt Disable flip-flops connected to it are set, thus disabling those devices from requesting interrupts. If the bit position contains a Ø, all Interrupt Disable flip-flops connected to it are reset, thus enabling those devices to request interrupts.

Because accumulator AC has 16 bit positions, there are 16 possible levels of interrupt priority.

Example:

A program is used for dedicated service as a controller for a lathe. However, it will permit only the teletype keyboard input to request an interrupt. Enable the interrupt request facility for this device. (Assume the TTI Interrupt Disable flip-flop is connected to data line 14 on the I/O bus).

```
     LDA      Ø, MASK
     MSKO  Ø          }      DOBS     Ø, CPU
     INTEN            }
     NIOS TTI
     - - - - - -

MASK: 177775      ;1/111/111/111/111/1Ø1
                    disables all devices but those connected to data line 14
                    on the I/O bus.
```

The I/O bus consists of a total of 47 lines. There are:

16 data lines for bi-directional data transfer.
6 device selection lines, decodable to 64 devices.
19 CPU → Device control lines (S, C, P, IORST, etc.)
6 Device → CPU control lines (BUSY, DONE, INTR, etc.)

INTERRUPT SERVICE ROUTINES

When all criteria exist for a device to request an interrupt, the device puts a logical one onto the INTR line. At the beginning of every memory cycle, the processor tests the INTR line for an interrupt request. If a request does exist, the processor resets the ION flag, thus disallowing any further interrupts, saves the contents of PC in location Ø and executes a JMP @1. Location 1 should contain a pointer to the Interrupt Service Routine.

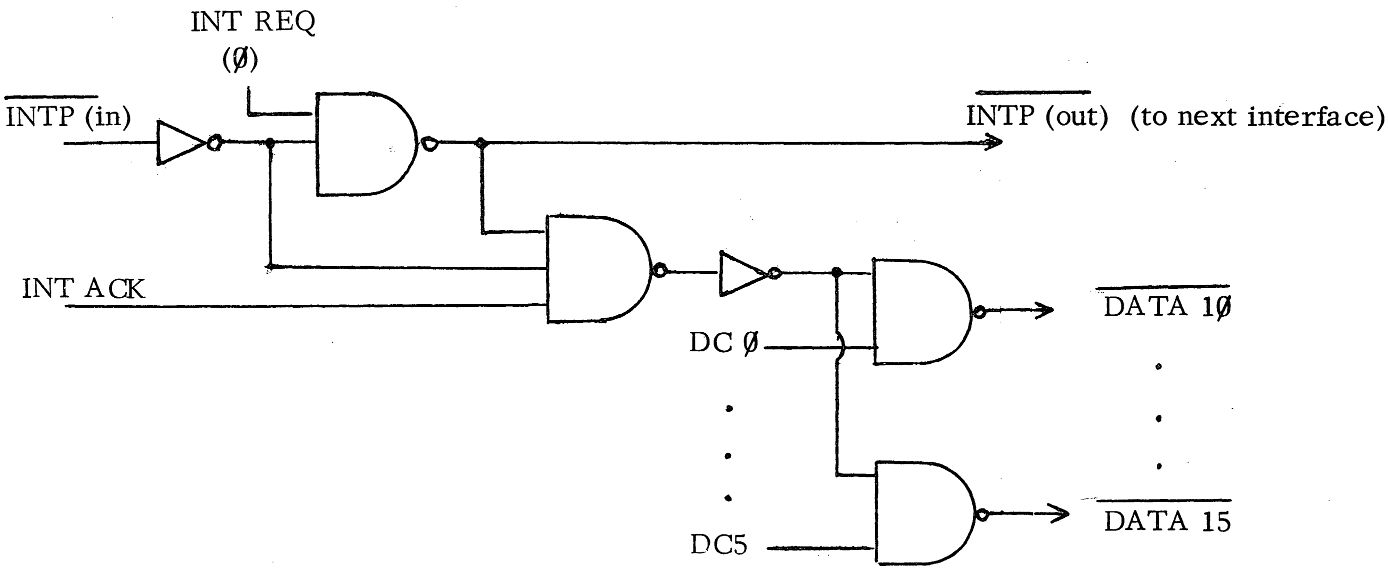
It is now up to the interrupt service routine to determine the device requesting the interrupt; to save the contents of any accumulators, Carry, or memory cells that may be destroyed by the service routine; to service the device; and then to restore the program to its state prior to the interrupt.

One method of determining the device requesting service is the straightforward polling technique. This technique involves simply checking the DONE flags of every device in descending order of priority.

Another method of determining the device requesting service is the broadcasting technique. When using this method, the interrupt service routine uses the command:

```
INTA    AC          ;INTERRUPT ACKNOWLEDGE
```

When this command is issued, the device requesting an interrupt which is physically closest (on the I/O bus) to the processor responds with its device code, and the device code is deposited to the AC. The hardware implementation of this action is:

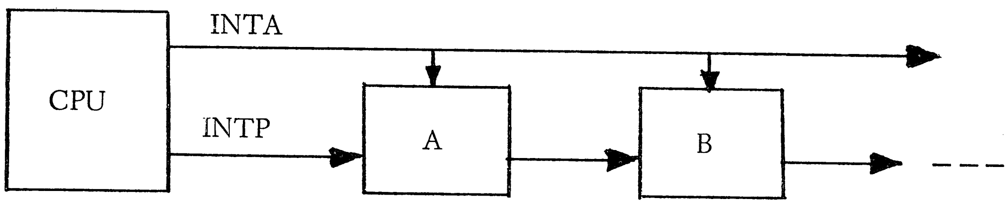


If the first device is requesting an interrupt, its Interrupt Request flip-flop is set (INT REQ(Ø) is at low level). This, in conjunction with an INTP IN of low level, results in INTP OUT being high and the device code of this device will be sent in response to a INTA. Only device code bits that are one need be sent. The device code is set using input levels DCØ to DC5, which are device dependent and are at high level for those bits to be 1. If the first device is not requesting an interrupt, INTP OUT remains low, and the device code is not sent.

Example:

If the PTP is the first device requesting an interrupt, INTP IN for the punch is low, INT REQ(Ø) is low, and therefore INTP OUT is high. All inputs to the 3-input nand gate in the figure are high, so the code is sent. For all prior device on the bus, INTP OUT was low, so the device codes were not sent. For all succeeding devices, the complement of INTP IN will be low, so their device codes will not be sent either.

The INTP OUT of this device is connected to INTP IN of the device with the next highest priority. This daisy chain effect appears as follows:



Thus, following INTA AC, accumulator AC bits 1Ø thru 15 contain the 6-bit device code of the first device on the bus that is requesting an interrupt.

Once the interrupt service routine has determined the device to be serviced, the routine can do one of two things: (1) service the device allowing no interrupts to occur during service; or (2) establish a new MSKO and then service the device, allowing higher priority devices to interrupt during service. If the second approach is employed, the contents of location Ø should first be saved, because it contains the re-entry pointer to the interrupted routine.

NOTE:

MSKO ≡ DOB AC, CPU

It is possible to activate the interrupt mechanism by using the expanded mnemonics:

DOB $\left\{\begin{matrix} S \\ C \end{matrix}\right\}$ AC, CPU

where: S = enable
C = disable

The command IORST (I/O RESET) resets all flip-flops in all devices.

POWER MONITOR AND AUTO-RESTART

The optional power monitor warns a program when power is failing by setting the Power Failure flag. If a system contains this option, the monitor will appear as any other I/O device to the interrupt system, except that it does not respond to an INTA command and must be serviced by:

	SKPDN	CPU
or	SKPDZ	CPU

The first function of the interrupt service routine should be to test this Power Failure flag. If this is the interrupting device, the program has 1 to 2 milliseconds to save the contents of the accumulators, Carry, and the contents of location Ø; to put a JMP to the desired restart location in location Ø; and then to HALT.

With the power switch in the LOCK position, when POWER UP occurs, the instruction in location Ø will be executed.

DATA CHANNEL

For devices requiring high transfer rates, direct memory access is provided by the Data Channel. Data channel commands have their own control lines on the I/O bus but use the same data lines for data transfer. The processor transfers data directly between the device and memory using only the memory buffer. No accumulators or other registers are used; thus, no saving of register contents is necessary when servicing a DCH request.

For I/O device - - - analogous - - -	for DCH device
INTR	DCHR
INTP	DCHP
INTA	DCHA

Instead of using a device selection code, a DCH device sets its DCH SEL (data channel select) flip-flop. A data channel I/O device informs the processor of the mode of data transfer it wants on the data channel mode lines DCHMØ, 1. The two I/O bus lines DCHMØ and DCHM1 select one of four transfer modes:

DCHMØ	DCHM1	
Ø	Ø	data out
1	Ø	data in
Ø	1	increment memory
1	1	add to memory (NOVA and SUPERNOVA only)

During increment memory and add to memory, the processor will give an output on the OVFLO (overflow) line of the I/O bus if the new contents of the memory cell in use exceed $2^{16}-1$.

CHAPTER 9

EXTENDED SOFTWARE

RELOCATABILITY

The use of relocation facilities allows the programmer to separately code, debug, and test subprograms without worrying about the absolute location of the program, or the absolute location of data and addresses shared by programs and subprograms at run time.

All subprograms are assembled with a relative starting address of Ø. Final address assignment is deferred until load time. The relocatable loader is then used to load these programs, assign absolute addresses, and define arguments that are being shared by or swapped from several programs.

It should be noted that absolute programs can still be written using the relocatable assembler since the relocatable assembler contains all of the features of the basic assembler plus some extensions. This means that programs that were originally assembled using the basic assembler need not be rewritten to be assembled on the relocatable assembler.

Relocatable coding can be of two types: zero relocatable and normal relocatable. Code written in the zero relocatable mode will reside in page zero when loaded. The pseudo-op indicating zero relocatable mode is .ZREL. Code written in the normal relocatable mode, indicated by the pseudo-op .NREL, may reside anywhere except page zero.

There are three pseudo-ops available for relocation mode assignments:

.LOC expression
.ZREL
.NREL

When the extended assembler is started, it is initially in the absolute mode. The assembler remains in a mode until it encounters one of the three relocation pseudo-ops. If the zero relocatable mode is entered via the pseudo-op .ZREL, succeeding labels are defined as zero relocatable mode symbols.

Another method of entering the zero relocatable mode is to use the .LOC expression pseudo-op, where expression contains a zero relocatable mode symbol. This method can be used once zero relocatable mode has been first entered with a .ZREL pseudo-op and one or more symbols have been defined in that mode.

The normal relocatable mode is handled in the same manner as the zero relocatable mode. It is entered by either of the following pseudo-ops: .NREL or .LOC expression, where the expression contains a previously defined normal relocatable mode symbol.

The absolute mode is entered with the pseudo-op .LOC expression, where the expression is either an octal number or contains an absolute mode symbol.

When loading a program using the relocatable loader, three location counters are used:

Absolute, .ZREL, .NREL .

The initial values of these counters are:

ABSOLUTE	Ø
.ZREL	5Ø
.NREL	4ØØ

When a mode is entered, each succeeding statement is assigned the address specified in its respective mode location counter; the mode location counter is then incremented by 1. A mode location counter may be incremented by more than one if, when in that mode, a .LOC .+ expression pseudo-op is used, where expression is an absolute expression.

As additional relocatably assembled programs are loaded, they are appended in memory to previously loaded programs. They are assigned the next available addresses as specified by the mode location counters.

The following statements contain examples of the use of relocation pseudo-ops.

```
ØØØØØ   ØØØØØØ           Ø                  ;ABSOLUTE
ØØØØ1   ØØØØØØ           Ø
        ØØØØ27           .LOC 27            ;ADJUST ABS LOC COUNTER
ØØØ27   ØØØ17Ø   A:      2*TABL
ØØØ3Ø   ØØØ113   B:      TABL+17
        ØØØØ74           .LOC .+43          ;ADJUST ABS LOC COUNTER
        ØØØØ2Ø   TABL:   .BLK 2Ø

                         .ZREL              ;ZERO RELOCATABLE
ØØØØØ-  ØØ351Ø   PNTR:   SUBRT
ØØØØ1-  ØØØØØØ'  PNTR1:  MAIN
        ØØØØØ7-          .LOC .+5           ;ADJUST ZREL LOC COUNTER
ØØØØ7-  ØØØØØØ   ARG1:   Ø

        ØØØ377           .LOC ARG1-PNTR+37Ø     ;ABSOLUTE

ØØ377   ØØØØØØ   ARG2:   Ø

                         .NREL              ;NORMAL RELOCATABLE
ØØØØØ'  Ø22Ø27   MAIN:   LDA Ø,@A
ØØØØ1'  Ø24Ø3Ø           LDA 1,B

        ØØØØ1Ø-          .LOC ARG1+1        ;ZERO RELOCATABLE
ØØØ1Ø-  Ø1ØØ74           ISZ TABL

        ØØ351Ø           .LOC 351Ø          ;ABSOLUTE AGAIN
Ø351Ø   Ø24ØØ7-  SUBRT:  LDA 1, ARG1
Ø3511   Ø3Ø377           LDA 2, ARG2

        ØØØØØ6'          .LOC MAIN+6        ;NORMAL RELOCATABLE
ØØØØ6'  Ø52Ø27           STA 2,@A

                         .END
```

INTERPROGRAM COMMUNICATION

In addition to assembling programs for loading by the relocatable loader, it is also possible, using the extended assembler, to produce subprograms that reference data, addresses, and constants that are defined in some other program.

Likewise, symbols defined in a program may be made available for referencing by other programs. It should be noted that, if a symbol is defined in one program and is made available for referencing, it should not also be defined and made available for referencing in another program since this would cause multiple symbol definition.

To define global symbols, which are symbols for use by external programs, an entry pseudo-op must be used:

.ENT $\underline{\text{symbol}}_1$, $\underline{\text{symbol}}_2$ ...

Other programs can now reference these symbols as externals. There are two types of externals: normal externals and displacement externals.

Displacement externals may be used in any MRI but must evaluate to 8 bits (Ø through 377_8). If a displacement external is to be used in a program, it must be declared as such with the pseudo-op:

.EXTD $\underline{\text{symbol}}_1$,$\underline{\text{symbol}}_2$ ...

Normal externals may be used in data statements only because they occupy an entire storage word. If a normal external is to be used in a program, it must be declared as such with the pseudo-op:

.EXTN $\underline{\text{symbol}}_1$, $\underline{\text{symbol}}_2$ ...

Every symbol declared as a displacement external or normal external in a subprogram should also be declared as an entry point in one of the other subprograms.

The pseudo-op:

.TITL <u>title</u>

defines the name of an assembled program.

The above four pseudo-ops must appear at the beginning of the subprogram, but they may occur in any order.

NOTE: It is good practice in using the Relocatable Loader to key mode 6 just prior to completing the load process. Mode 6 lists all global symbols and their values. The starting address can easily be found if it is labeled as a global symbol.

INTERPROGRAM COMMUNICATION (Continued)

The following two pages show two communicating programs, REPUS and AVON and the pseudo-ops that permit interprogram referencing.

```
                                   .TITL          REPUS

                                   .ENT           BGN,CCRLF,.CRLF
                                   .EXTN          CRLF,TYPET
                                   .EXTD          C377,DONE

                                   .ZREL
ØØØØØ-  ØØ1764"  CSTR:             STRING+STRING
ØØØØ1-  ØØ1776"  CSTR1:            STRING+STRING+12
        ØØØØØ1   PNTR:             .BLK 1
ØØØØ3-  177777   .CRLF:            CRLF
ØØØØ4-  ØØ5Ø15   CCRLF:            5Ø15

                                   .NREL
ØØØØØ'  ØØ6ØØ3-  BGN:              JSR @.CRLF
ØØØØ1'  Ø2ØØØ1-                    LDA Ø,CSTR1
ØØØØ2'  Ø4ØØØ2-                    STA Ø,PNTR
ØØØØ3'  Ø3ØØØ2-  LOOP:             LDA 2,PNTR
ØØØØ4'  Ø14ØØ2-                    DSZ PNTR
ØØØØ5'  Ø24ØØØ-                    LDA 1,CSTR
ØØØØ6'  132433                     SUBZ# 1,2,SNC
ØØØØ7'  ØØØØØ2$                    JMP DONE
ØØØ1Ø'  15122Ø                     MOVZR 2,2
ØØØ11'  Ø21ØØØ                     LDA Ø,Ø,2
ØØØ12'  Ø24ØØ1$                    LDA 1,C377
ØØØ13'  1Ø1ØØ2                     MOV Ø,Ø,SZC
ØØØ14'  1Ø13ØØ                     MOVS Ø,Ø
ØØØ15'  1234ØØ                     AND 1,Ø
ØØØ16'  177777                     TYPET
ØØØ17'  ØØØ764                     JMP LOOP
ØØØ2Ø'  Ø6Ø111   INIT:             NIOS TTO
ØØØ21'  ØØØ757                     JMP BGN
        ØØØ772'                    .LOC .+75Ø
ØØ772'  Ø4Ø44Ø   STRING:           .TXT * A
ØØ773'  Ø47526   VO
ØØ774'  Ø2Ø116   N
ØØ775'  Ø42522   RE
ØØ776'  Ø5252Ø   PU
ØØ777'  Ø2Ø123   S
Ø1ØØØ'  ØØØØØØ   *

        ØØØØ2Ø'                    .END INIT
```

```
                                  .TITL    AVON
                                  .EXTD    CCRLF,.CRLF
                                  .EXTN    BGN
                                  .ENT     C377,DONE, TYPET,CRLF

                                  .ZREL
ØØØØØ-   ØØØ377   C377:          377
0ØØØ1-   ØØØØØ7'  .TTTO:         TTTO
0ØØØ2-   177777   .BGN:          BGN
ØØØØ3-   ØØ6ØØ2$  DONE:          JSR @.CRLF
ØØØØ4-   Ø63Ø77                  HALT
ØØØ05-   ØØ2ØØ2-                 JMP @.BGN
         ØØ6ØØ1-  TYPET=         JSR @.TTTO

                                  .NREL
ØØØØØ'   Ø544Ø6   CRLF:          STA 3,RCRLF
ØØØØ1'   Ø2ØØØ1$                 LDA Ø,CCRLF
ØØØØ2'   ØØ6ØØ1-                 TYPET
ØØØØ3'   1Ø13ØØ                  MOVS Ø,Ø
ØØØØ4'   ØØ6ØØ1-                 TYPET
ØØØØ5'   ØØ24Ø1                  JMP @RCRLF
         ØØØØØ1   RCRLF:         .BLK 1
ØØØØ7'   Ø63611   TTTO:          SKPDN TTO
ØØØ1Ø'   ØØØ777                  JMP .-1
ØØØ11'   Ø61111                  DOAS Ø,TTO
ØØØ12'   ØØ14ØØ                  JMP Ø,3
                                  .END
```

ASSEMBLER EXTENSIONS

In addition to handling relocatable programming and interprogram communications, the extended assembler has expanded number definition capabilities.

Radix declarations using the .RDX pseudo-op have been expanded to allow decimal numbers to be input at any point in the program, under any radix mode, by simply following the number with a decimal point.

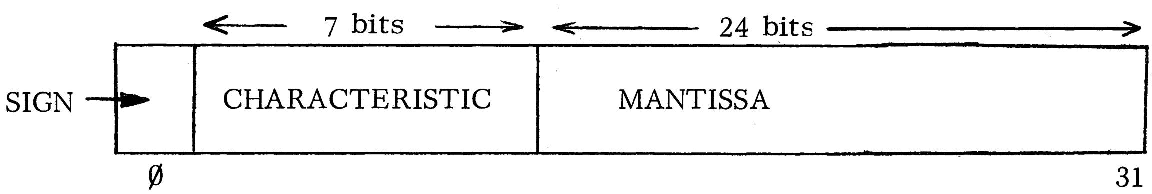
Example:

.RDX 3			Decimal point after 1Ø8 indicates base 1Ø; all numbers following the pseudo-op that do not have a decimal point are interpreted as base 3.
1Ø8.	=	$1Ø8_{1Ø}$	
1Ø2	=	$1Ø2_{3}$	

Floating Point Numbers

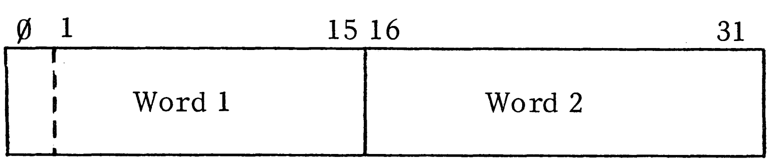
If the decimal point is followed by one or more numerals, or if a number is followed by the letter E, the number is interpreted as a floating point number to be used with the Floating Point Interpreter package.

Floating point numbers may only be used in data statements, are two words long, and have the following format:



Double Precision Numbers

If a number is followed by the letter D, it is handled as a double precision integer. That is, it is stored in two memory words as follows:



The number is represented in two's complement form.

Example:

12Ø7ØØØØD

.RDX 5

134Ø2D ⇒ 2116_8 ⇒

Ø	Ø	Ø	Ø	5	Ø	Ø	7	Ø	Ø	Ø	Ø
Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø	2	1	1	6

If a decimal point separates the number and the letter D, the number is handled as a double precision decimal number.

Example:

1ØØØØØ.D ⇒ $3Ø324Ø_8$

⇒

Ø	Ø	Ø	Ø	Ø	1	1	Ø	3	2	4	Ø

Double precision numbers may only be used in data statements.

Bit Boundary Alignment

Binary equivalents of integers may be aligned within a 16-bit word by the Bit Boundary Alignment technique. The statement takes the form:

number B decimal number

Example:

number

1Ø3B1Ø = Ø ØØØ 1ØØ ØØ1 1ØØ ØØØ

bit position 1Ø

Example:

.RDX 5
134Ø2B14

134Ø2 ⇒ Ø ØØØ Ø1Ø ØØ1 ØØ1 11Ø

right justify to bit position 14

Ø ØØØ 1ØØ Ø1Ø Ø11 1ØØ

Example:

A data word has 3 fields:

W 1 (5)	W 2 (7)	W3 (4)
Ø – 4	5 – 11	12 – 15

The mask for W2 is 3760

or (# of bits) B (alignment of rightmost bit) ⇒ 177B11

The mask for W1 is 37B4

the mask for W1+W3 is 37B4+17B15 or 37B4+17

Conditional Assembly

The extended assembler allows portions of a program to be by-passed or assembled at assembly time. This conditional assembly feature is controlled by the 3 pseudo-ops:

.IFE absolute expression

or .IFN absolute expression

and .ENDC

A portion of a program occurring between an .IFE or .IFN pseudo-op and an .ENDC pseudo-op is or is not assembled based on the evaluation of the absolute expression.

```
For the format  .IFE    absolute expression
                ------
                ------
                ------
                .ENDC
```

the program between the pseudo-ops will be assembled if the expression evaluates to Ø. The program section will be by-passed for a non-Ø expression evaluation.

```
For the format  .IFN    absolute expression
                -----
                -----
                -----
                .ENDC
```

non-Ø evaluation causes assembly and Ø evaluation causes program by-pass.

Conditional assembly might be used to remove I/O drive routines from a very general program if certain peripheral devices are not used. Conditional assembly blocks cannot be nested or overlapped, but they may contain .EOT and/or .END pseudo-ops.

SYMBOLIC DEBUGGER

If the symbolic debugging features of Debug III are to be used to debug the program, the Pass 2 BIN: mode of the extended assembler must include punching of local symbols. Use assembler mode 3 or 4 so local symbols will be punched for use by Debug III. These symbols must also be loaded, so the first mode command to the relocatable loader should be 4. (See page 3-12 for assembly modes).

In any case, global symbols (those symbols declared as entry points) will always be punched, loaded, and recognizable.

CHAPTER 10

BYTE MANIPULATION

In many applications, 8-bit words -- bytes -- are sufficient data word blocks, such as for storage of 8-bit teletype character strings.

Because the address of any 16-bit word requires only 15 bits, the remaining bit can be used to specify the left or right byte of the contents of a memory location.

A memory capacity of 32K words contains 64K bytes, where each memory cell contains 2 bytes.

Ø 7	8 15
B	A

It has been seen how word addresses or word pointers are of the format:

Ø	1 15
I	15-bit word address

Similarly, byte addresses or byte pointers are of the form:

Ø 14	15
15-bit word address	B

where B= Ø specifies the right byte (A)
B= 1 specifies the left byte (B)

Thus, incrementing the byte pointer addresses first the right byte and then the left byte of sequential memory locations.

Right shifting the byte pointer leaves a memory address. Following this with program skipping based on the Carry flag designates the specific byte.

CHAPTER 11

PROGRAMMING TRICKS

1. Clear AC and Carry.

```
SUBO    AC,AC
```

2. Clear AC and preserve Carry.

```
SUBC    AC,AC
```

3. Generate the indicated constants.

```
SUBZL   AC,AC       ;generate +1
ADC     AC,AC       ;generate -1
ADCZL   AC,AC       ;generate -2
```

4. Let ACX be any accumulator <u>whose contents are zero</u>. Let ACY be any other accumulator. Generate the indicated constants in ACY.

```
INCZL   ACX,ACY   ;generate +2
INCOL   ACX,ACY   ;generate +3
INCS    ACX,ACY   ;generate +4ØØ8
```

5. Subtract 1 from an accumulator without using a constant from memory.

```
NEG     AC,AC
COM     AC,AC
```

6. Check if both bytes in an accumulator are equal.

```
MOVS    ACS,ACD
SUB     ACS,ACD,SZR
JMP     ---         ;not equal
---     ---         ;equal
```

7. Check if two accumulators are both zero.

```
MOV     ACS,ACS,SNR
SUB     ACS,ACD,SZR
JMP     ---         ;not both zero
---     ---         ;both zero
```

8. Check an ASCII character to make sure it is a decimal digit. The character is in ACS and is not destroyed by the test. Accumulators ACX and ACY are destroyed.

```
        LDA     ACX,C6Ø              ;ACX=ASCII zero
        LDA     ACY,C71              ;ACY=ASCII nine
        ADCZ#   ACY,ACS,SNC          ;skips if (ACS) > 9
        ADCZ#   ACS,ACX,SZC          ;skips if (ACS) ≥ Ø
        JMP     ---                  ;not digit
        ---     ---                  ;digit

        C6Ø:    6Ø                   ;ASCII zero
        C71:    71                   ;ASCII nine
```

9. Test an accumulator for zero.

```
        MOV     AC,AC,SZR
        JMP     ---                  ;not zero
        ---     ---                  ;zero
```

10. Test an accumulator for -1.

```
        COM#    AC,AC,SZR
        JMP     ---                  ;not -1
        ---     ---                  ; -1
```

11. Test an accumulator for 2 or greater.

```
        MOVZR#  AC,AC,SNR
        JMP     ---                  ;less than 2
        ---     ---                  ;2 or greater
```

12. Assume it is known that AC contains Ø, 1, 2, or 3. Find out which one.

```
        MOVZR#  AC,AC,SEZ
        JMP     THREE                ;was 3
        MOV     AC,AC,SNR
        JMP     ZERO                 ;was Ø
        MOVZR#  AC,AC,SZR
        JMP     TWO                  ;was 2
        ---     ---                  ;was 1
```

13. Multiply an AC by the indicated value.

```
MOV     ACX,ACX     ;multiply by 1

MOVZL   ACX,ACX     ;multiply by 2

MOVZL   ACX,ACY     ;multiply by 3
ADD     ACY,ACX

ADDZL   ACX,ACX     ;multiply by 4

MOV     ACX,ACY     ;multiply by 5
ADDZL   ACX,ACX
ADD     ACY,ACX

MOVZL   ACX,ACY     ;multiply by 6
ADDZL   ACY,ACX

MOVZL   ACX,ACY     ;multiply by 7
ADDZL   ACY,ACY
SUB     ACX,ACY     ;in ACY

ADDZL   ACX,ACX     ;multiply by 8
MOVZL   ACX,ACX

MOVZL   ACX,ACY     ;multiply by 9
ADDZL   ACY,ACY
ADD     ACY,ACX

MOV     ACX,ACY     ;multiply by 10(10)
ADDZL   ACX,ACX
ADDZL   ACY,ACX

MOVZL   ACX,ACY     ;multiply by 12(10)
ADDZL   ACY,ACX
MOVZL   ACX,ACX

MOVZL   ACX,ACY     ;multiply by 18(10)
ADDZL   ACY,ACY
ADDZL   ACY,ACX
```

APPENDIX A

SAMPLE PROGRAMS

Following are some programs that illustrate some of the features described in this document.

```
                    ;ROUTINE TO READ CHARACTERS FROM THE TELETYPE.  AS
                    ;EACH CHARACTER IS READ, IT IS ECHOED.  IF A CARRIAGE
                    ;RETURN CHARACTER IS INPUT, THE ROUTINE AUTOMATICALLY
                    ;GENERATES A LINE FEED.

                    ;CALLING SEQUENCE:
                    ;       JSR     GETC            USED WITH PUTC ON PAGE A-3.

                    ;OUTPUT:
                    ;               ACØ=CHARACTER RIGHT JUSTIFIED
      ØØØ4ØØ                    .LOC    4ØØ

ØØ4ØØ Ø54433 GETC:              STA     3,SGET     ;SAVE RETURN ADDRESS
ØØ4Ø1 Ø6Ø11Ø                    NIOS    TTI        ;START TELETYPE
ØØ4Ø2 Ø6361Ø                    SKPDN   TTI        ;WAIT FOR INPUT
ØØ4Ø3 ØØØ777                    JMP     .-1
ØØ4Ø4 Ø6Ø61Ø                    DIAC    Ø,TTI      ;GET CHAR AND CLEAR TTY
ØØ4Ø5 Ø24431                    LDA     1,MSK      ;AC1=177
ØØ4Ø6 1234ØØ                    AND     1,Ø        ;KEEP RIGHT 7 BITS
ØØ4Ø7 ØØ441Ø                    JSR     PUTC       ;OUTPUT CHARACTER
ØØ41Ø Ø34425                    LDA     3,CR       ;CHECK FOR CR
ØØ411 1164Ø4                    SUB     Ø,3,SZR    ;SKIP IF CR
ØØ412 ØØ2421                    JMP     @SGET      ;NOT CR-RETURN
ØØ413 Ø2Ø424                    LDA     Ø,LF       ;CR-GENERATE LF
ØØ414 ØØ44Ø3                    JSR     PUTC
00415 Ø2Ø42Ø                    LDA     Ø,CR       ;RESTORE CR
ØØ416 ØØ2415                    JMP     @SGET      ;RETURN
```

```
;ROUTINE TO OUTPUT CHARACTERS ON THE TELETYPE.  IF
;THE CHARACTER OUTPUT IS NULL, THE ROUTINE WILL
;AUTOMATICALLY GENERATE A CR AND LF.

;CALLING SEQUENCE:

;               JSR         PUTC             USED WITH GETC ON
                                             PAGE A-2
;INPUT:
;               ACØ=CHARACTER RIGHT JUSTIFIED

ØØ417 Ø63511 PUTC:   SKPBZ   TTO        ;WAIT UNTIL NOT BUSY
ØØ42Ø ØØØ777         JMP     .-1
ØØ421 Ø61111         DOAS    Ø,TTO      ;OUTPUT CHARACTER
ØØ422 1Ø1ØØ4         MOV     Ø,Ø,SZR    ;SKIP IF NULL CHAR
ØØ423 ØØ14ØØ         JMP     Ø,3        ;NULL-RETURN
ØØ424 Ø5441Ø         STA     3,SPUT     ;SAVE RETURN
ØØ425 Ø2Ø41Ø         LDA     Ø,CR       ;OUTPUT CR
ØØ426 ØØ4771         JSR     PUTC       ;RECURSE
ØØ427 Ø2Ø41Ø         LDA     Ø,LF       ;OUTPUT LF
ØØ43Ø ØØ4767         JSR     PUTC       ;RECURSE
ØØ431 1Ø24ØØ         SUB     Ø,Ø        ;RESTORE NULL
ØØ432 ØØ24Ø2         JMP     @SPUT      ;NOT NULL - RETURN

ØØ433 ØØØØØØ SGET:   Ø                  ;SAVE FOR RETURN (GETC)
ØØ434 ØØØØØØ SPUT:   Ø                  ;SAVE FOR RETURN (PUTC)
ØØ435 ØØØØ15 CR:     15                 ;ASCII CARRIAGE RETURN
ØØ436 ØØØ177 MSK:    177                ;MASK FOR RIGHT 7 BITS
ØØ437 ØØØØ12 LF:     12                 ;ASCII LINE FEED
```

```
                ;OCTAL TO BINARY CONVERSION ROUTINE.  THE ROUTINE
                ;CONVERTS UNSIGNED OCTAL NUMBERS TO BINARY.  THE
                ;ROUTINE CONTINUES TO INPUT NUMBERS UNTIL IT SEES A
                ;CARRIAGE RETURN WHICH SIGNALS THE END OF THE NUMBER.

                ;CALLING SEQUENCE:
                ;               JSR     OCTBN

                ;INPUT:
                ;               THIS ROUTINE ASSUMES AN INPUT ROUTINE IS
                ;               AVAILABLE AND IS POINTED TO BY LOCATION
                ;               4Ø IN PAGE ZERO.  WHENEVER THIS ROUTINE
                ;               NEEDS A CHARACTER, IT DOES AN INDIRECT
                ;               JUMP THRU LOCATION 4Ø.

                ;OUTPUT:
                ;               AC1=CONVERTED OCTAL NUMBER.

      ØØØØ4Ø    AGET=4Ø

ØØØØØ Ø54Ø14    OCTBN:  STA     3,SAVE          ;SAVE RETURN ADDRESS
ØØØØ1 1264ØØ            SUB     1,1             ;CLEAR AC1
ØØØØ2 ØØ6Ø4Ø    OCTL:   JSR     @AGET           ;GET A CHARACTER
ØØØØ3 Ø34Ø15            LDA     3,CR            ;AC3=ASCII CR
ØØØØ4 116415            SUB#    Ø,3,SNR         ;SKIP IF NOT CR
ØØØØ5 ØØ2Ø14            JMP     @SAVE           ;CR-RETURN
ØØØØ6 Ø34Ø16            LDA     3,C7            ;AC3=MASK FOR 3 BITS
ØØØØ7 1634ØØ            AND     3,Ø             ;ACØ=RIGHTMOST 3 BITS
ØØØ1Ø 12712Ø            ADDZL   1,1             ;SHIFT AC1 LEFT 3
ØØØ11 12512Ø            MOVZL   1,1
ØØØ12 1Ø7ØØØ            ADD     Ø,1             ;ADD IN NEW BITS
ØØØ13 ØØØØØ2            JMP     OCTL            ;LOOP

ØØØ14 ØØØØØØ    SAVE:   Ø                       ;SAVE FOR RETURN ADDRESS
ØØØ15 ØØØØ15    CR:     15                      ;ASCII CARRIAGE RETURN
ØØØ16 ØØØØØ7    C7:     7                       ;MASK FOR 3 BITS

                        .END
```

```
;BINARY TO OCTAL CONVERSION ROUTINE.  THE ROUTINE
;CONVERTS A 16-BIT BINARY INTEGER TO AN ASCII
;CHARACTER STRING FOR OUTPUT.

;CALLING SEQUENCE:
;             JSR      BNOCT

;INPUT:
;             AC1=INTEGER TO BE CONVERTED

;OUTPUT:
;             THIS ROUTINE ASSUMES AN OUTPUT ROUTINE IS
;             AVAILABLE AND IS POINTED TO BY LOCATION 41
;             IN PAGE ZERO.  AS EACH ASCII CHARACTER IS
;             GENERATED, THIS ROUTINE DOES AN INDIRECT
;             JSR THRU LOCATION 41.

      ØØØØ41  APUT=41

ØØØØØ Ø54Ø16  BNOCT:  STA    3,SAVE          ;SAVE RETURN ADDRESS
ØØØØ1 15262Ø          SUBZR   2,2            ;AC2=1ØØØØØ
ØØØØ2 Ø2ØØ15  LOOP:   LDA    Ø,C6Ø           ;ACØ=ASCII ZERO
ØØØØ3 146443          SUBO   2,1,SNC         ;STILL PLUS IF NO CARRY
ØØØØ4 1Ø14Ø1          INC    Ø,Ø,SKP         ;INC ASCII CHAR
ØØØØ5 147ØØ1          ADD    2,1,SKP         ;TOO MUCH-ADD BACK
ØØØØ6 ØØØØØ3          JMP    .-3
ØØØØ7 ØØ6Ø41          JSR    @APUT           ;OUTPUT CHARACTER
ØØØ1Ø 15122Ø          MOVZR  2,2             ;SHIFT ONE BIT RIGHT
ØØØ11 15122Ø          MOVZR  2,2
ØØØ12 151224          MOVZR  2,2,SZR         ;LAST DIGIT?
ØØØ13 ØØØØØ2          JMP    LOOP            ;NO-CONTINUE
ØØØ14 ØØ2Ø16          JMP    @SAVE           ;YES-RETURN

ØØØ15 ØØØØ6Ø  C6Ø:    6Ø                     ;ASCII ZERO
ØØØ16 ØØØØØØ  SAVE:   Ø                      ;SAVE FOR RETURN ADDRESS

                      .END
```

```
                    ;BINARY TO DECIMAL CONVERSION ROUTINE.  THE ROUTINE
                    ;CONVERTS A 16-BIT UNSIGNED BINARY INTEGER TO
                    ;AN ASCII CHARACTER STRING FOR OUTPUT.  IT DOES
                    ;NOT SUPPRESS LEADING ZEROES.

                    ;CALLING SEQUENCE:
                    ;          JSR     BNDEC

                    ;INPUT:
                    ;          AC1=BINARY INTEGER TO BE CONVERTED.

                    ;OUTPUT:
                    ;          THIS ROUTINE ASSUMES AN OUTPUT ROUTINE IS
                    ;          AVAILABLE AND IS POINTED TO BY LOCATION
                    ;          41 IN PAGE ZERO.  AS EACH CHARACTER IS
                    ;          GENERATED, THIS ROUTINE DOES AN INDIRECT
                    ;          JSR THRU LOCATION 41.

      ØØØØ41        APUT=41

      ØØØ4ØØ                  .LOC    4ØØ
ØØ4ØØ Ø54425        BNDEC:    STA     3,SAVE        ;SAVE ROUTINE ADDRESS
ØØ4Ø1 Ø34422                  LDA     3,INST        ;SET UP LDA COMMAND
ØØ4Ø2 Ø544Ø1                  STA     3,.+1
ØØ4Ø3 ØØØØØØ        LOOP:     Ø                     ;AC2=POWER OF 1Ø
ØØ4Ø4 Ø2Ø42Ø                  LDA     Ø,C6Ø         ;ACØ=ASCII ZERO
ØØ4Ø5 146443                  SUBO    2,1,SNC       ;STILL PLUS IF NO CARRY
ØØ4Ø6 1Ø14Ø1                  INC     Ø,Ø,SKP       ;INC ASCII CHAR
ØØ4Ø7 147ØØ1                  ADD     2,1,SKP       ;TOO MUCH-ADD BACK
ØØ41Ø ØØØ775                  JMP     .-3
ØØ411 ØØ6Ø41                  JSR     @APUT         ;OUTPUT CHARACTER
ØØ412 Ø1Ø771                  ISZ     LOOP          ;INC LDA COMMAND
ØØ413 1512Ø3                  MOVR    2,2,SNC       ;LAST DIGIT?
ØØ414 ØØØ767                  JMP     LOOP          ;NO-CONTINUE
ØØ415 ØØ241Ø                  JMP     @SAVE         ;YES-RETURN

      ØØØØ12                  .RDX    1Ø            ;CHANGE RADIX TO 1Ø
ØØ416 Ø2342Ø        TENS:     1ØØØØ
ØØ417 ØØ175Ø                  1ØØØ
ØØ42Ø ØØØ144                  1ØØ
ØØ421 ØØØØ12                  1Ø
ØØ422 ØØØØØ1                  1
      ØØØØ1Ø                  .RDX    8             ;CHANGE BACK TO 8

ØØ423 Ø3Ø413        INST:     LDA     2,.+TENS-LOOP
ØØ424 ØØØØ6Ø        C6Ø:      6Ø                    ;ASCII ZERO
ØØ425 ØØØØØØ        SAVE:     Ø                     ;SAVE FOR RETURN

                              .END
```

```
            ØØØ4ØØ      .LOC 4ØØ
                                   ;DUMP PROGRAM

ØØ4ØØ 1Ø24ØØ  TA:     SUB   Ø,Ø          ;ACØ=NULL
ØØ4Ø1 ØØ4534          JSR   PUTC         ;OUTPUT CR-LF
ØØ4Ø2 Ø2Ø446          LDA   Ø,L          ;OUTPUT L
ØØ4Ø3 ØØ4532          JSR   PUTC
ØØ4Ø4 ØØ4435          JSR   TC           ;OUTPUT B=
ØØ4Ø5 ØØ4454          JSR   OCTBN        ;READ LB
ØØ4Ø6 Ø5Ø447          STA   2,LB         ;SAVE LB
ØØ4Ø7 Ø2Ø442          LDA   Ø,U          ;OUTPUT U
ØØ41Ø ØØ4525          JSR   PUTC
ØØ411 ØØ443Ø          JSR   TC           ;OUTPUT B=
ØØ412 ØØ4447          JSR   OCTBN        ;READ UB
ØØ413 Ø5Ø443          STA   2,UB         ;SAVE UB
ØØ414 Ø1Ø442          ISZ   UB
ØØ415 Ø24442  TB:     LDA   1,K          ;SET COUNTER
ØØ416 Ø44442          STA   1,C
ØØ417 Ø24436          LDA   1,LB         ;OUTPUTADDRESS
ØØ42Ø ØØ446Ø          JSR   BNOCT
ØØ421 Ø2Ø434  TD:     LDA   Ø,LB         ;COMPARE LB AND UB
ØØ422 Ø24434          LDA   1,UB
ØØ423 1Ø6415          SUB#  Ø,1,SNR
ØØ424 ØØØ754          JMP   TA           ;EQUAL-RESTART
ØØ425 Ø2Ø427          LDA   Ø,SP         ;UNEQUAL-OUTPUT SPACE
ØØ426 ØØ45Ø7          JSR   PUTC
ØØ427 Ø26426          LDA   @1,LB        ;OUTPUT (LB)
ØØ43Ø ØØ445Ø          JSR   BNOCT
ØØ431 Ø1Ø424          ISZ   LB           ;INCR LB
ØØ432 ØØØ4Ø2          JMP   .+2
ØØ433 ØØØ745          JMP   TA           ;RESTART
ØØ434 Ø14424          DSZ   C            ;DECR COUNTER
ØØ435 ØØØ764          JMP   TD           ;CONTINUE
ØØ436 1Ø24ØØ          SUB   Ø,Ø          ;ACØ=NULL
ØØ437 ØØ4476          JSR   PUTC         ;OUTPUT CR-LF
ØØ44Ø ØØØ755          JMP   TB           ;NEW LINE
ØØ441 Ø544Ø6  TC:     STA   3,STC        ;SAVE RETURN
ØØ442 Ø2Ø41Ø          LDA   Ø,B          ;OUTPUT B
ØØ443 ØØ4472          JSR   PUTC
ØØ444 Ø2Ø4Ø7          LDA   Ø,EQ         ;OUTPUT=
ØØ445 ØØ447Ø          JSR   PUTC
ØØ446 ØØ24Ø1          JMP   @STC         ;RETURN
ØØ447 ØØØØØØ  STC:    Ø                  ;SAVE FOR RETURN (TC)
ØØ45Ø ØØØ114  L:      "L
ØØ451 ØØØ125  U:      "U
ØØ452 ØØØ1Ø2  B:      "B
ØØ453 ØØØØ75  EQ:     "=
ØØ454 ØØØØ4Ø  SP:     "
ØØ455 ØØØØØØ  LB:     Ø                  ;LOWER BOUND
ØØ456 ØØØØØØ  UB:     Ø                  ;UPPER BOUND
ØØ457 ØØØØ1Ø  K:      1Ø                 ;CONSTANT
ØØ46Ø ØØØØØØ  C:      Ø                  ;COUNTER
```

```
ØØ461 Ø54414   OCTBN:   STA     3,SAVE
ØØ462 1524ØØ            SUB     2,2
ØØ463 ØØ4433   OCT1:    JSR     GETC
ØØ464 Ø34413            LDA     3,CR
ØØ465 116415            SUB#    Ø,3,SNR
ØØ466 ØØ2407            JMP     @SAVE
ØØ467 Ø34407            LDA     3,C7
ØØ47Ø 1634ØØ            AND     3,Ø
ØØ471 15312Ø            ADDZL   2,2
ØØ472 15112Ø            MOVZL   2,2
ØØ473 113ØØØ            ADD     Ø,2
ØØ474 ØØØ767            JMP     OCT1

ØØ475 ØØØØØØ   SAVE:    Ø
ØØ476 ØØØØØ7   C7:      7
ØØ477 ØØØØ15   CR:      15

ØØ5ØØ Ø54775   BNOCT:   STA     3,SAVE
ØØ5Ø1 15262Ø            SUBZR   2,2
ØØ5Ø2 Ø2Ø413   LOOP:    LDA     Ø,C6Ø
ØØ5Ø3 146443            SUBO    2,1,SNC
ØØ5Ø4 1Ø14Ø1            INC     Ø,Ø,SKP
ØØ5Ø5 147ØØ1            ADD     2,1,SKP
ØØ5Ø6 ØØØ775            JMP     .-3
ØØ5Ø7 ØØ4426            JSR     PUTC
ØØ51Ø 15122Ø            MOVZR   2,2
ØØ511 15122Ø            MOVZR   2,2
ØØ512 151224            MOVZR   2,2,SZR
ØØ513 ØØØ767            JMP     LOOP
ØØ514 ØØ2761            JMP     @SAVE

ØØ515 ØØØØ6Ø   C6Ø:     6Ø

ØØ516 Ø54433   GETC:    STA     3,SGET
ØØ517 Ø6Ø11Ø            NIOS    TTI
ØØ52Ø Ø6361Ø            SKPDN   TTI
ØØ521 ØØØ777            JMP     .-1
ØØ522 Ø6Ø61Ø            DIAC    Ø,TTI
ØØ523 Ø2443Ø            LDA     1,MSK
ØØ524 1234ØØ            AND     1,Ø
ØØ525 ØØ441Ø            JSR     PUTC
ØØ526 Ø34751            LDA     3,CR
ØØ527 1164Ø4            SUB     Ø,3,SZR
ØØ53Ø ØØ2421            JMP     @SGET
ØØ531 Ø2Ø423            LDA     Ø,LF
ØØ532 ØØ44Ø3            JSR     PUTC
ØØ533 Ø2Ø744            LDA     Ø,CR
ØØ534 ØØ2415            JMP     @SGET
```

```
ØØ535 Ø63511   PUTC:    SKPBZ   TTO
ØØ536 ØØØ777            JMP     .-1
ØØ537 Ø61111            DOAS    Ø,TTO
ØØ54Ø 1Ø1ØØ4            MOV     Ø,Ø,SZR
ØØ541 ØØ14ØØ            JMP     Ø,3
ØØ542 Ø5441Ø            STA     3,SPUT
ØØ543 Ø2Ø734            LDA     Ø,CR
ØØ544 ØØ4771            JSR     PUTC
ØØ545 Ø2Ø4Ø7            LDA     Ø,LF
ØØ546 ØØ4767            JSR     PUTC

ØØ547 1Ø24ØØ            SUB     Ø,Ø
ØØ55Ø ØØ24Ø2            JMP     @SPUT

ØØ551 ØØØØØØ   SGET:    Ø
ØØ552 ØØØØØØ   SPUT:    Ø
ØØ553 ØØØ177   MSK:     177
ØØ554 ØØØØ12   LF:      12

      ØØØØ4Ø            .LOC    4Ø
ØØØ4Ø ØØØ516            GETC
ØØØ41 ØØØ535            PUTC

                        .END
```

```
                          .TITL   DUMP
                          .EXTD   APUTC, AOCBN, ABNOC
                          .NREL

ØØØØØ'1Ø24ØØ    TA:       SUB     Ø,Ø
ØØØØ1'ØØ6ØØ1$             JSR     @APUTC
ØØØØ2'Ø2Ø446              LDA     Ø, L
ØØØØ3'ØØ6ØØ1$             JSR     @APUTC
ØØØØ4'ØØ4435              JSR     TC
ØØØØ5'ØØ6ØØ2$             JSR     @AOCBN
ØØØØ6'Ø5Ø447              STA     2, LB
ØØØØ7'Ø2Ø442              LDA     Ø, U
ØØØ1Ø'ØØ6ØØ1$             JSR     @APUTC
ØØØ11'ØØ443Ø              JSR     TC
ØØØ12'ØØ6ØØ2$             JSR     @AOCBN
ØØØ13'Ø5Ø443              STA     2, UB
ØØØ14'Ø1Ø442              ISZ     UB
ØØØ15'Ø24442    TB:       LDA     1, K
ØØØ16'Ø44442              STA     1, C
ØØØ17'Ø24436              LDA     1, LB
ØØØ2Ø'ØØ6ØØ3$             JSR     @ABNOC
ØØØ21'Ø2Ø434    TD:       LDA     Ø, LB
ØØØ22'Ø24434              LDA     1, UB
ØØØ23'1Ø6415              SUB#    Ø, 1, SNR
ØØØ24'ØØØ754              JMP     TA
ØØØ25'Ø2Ø427              LDA     Ø, SP
ØØØ26'ØØ6ØØ1$             JSR     @APUTC
ØØØ27'Ø26426              LDA     @1, LB
ØØØ3Ø'ØØ6ØØ3$             JSR     @ABNOC
ØØØ31'Ø1Ø424              ISZ     LB
ØØØ32'ØØØ4Ø2              JMP     .+2
ØØØ33'ØØØ745              JMP     TA
ØØØ34'Ø14424              DSZ     C
ØØØ35'ØØØ764              JMP     TD
ØØØ36'1Ø24ØØ              SUB     Ø,Ø
ØØØ37'ØØ6ØØ1$             JSR     @APUTC
ØØØ4Ø'ØØØ755              JMP     TB
ØØØ41'Ø544Ø6    TC:       STA     3, STC
ØØØ42'Ø2Ø41Ø              LDA     Ø, B
ØØØ43'ØØ6ØØ1$             JSR     @APUTC
ØØØ44'Ø2Ø4Ø7              LDA     Ø, EQ
ØØØ45'ØØ6ØØ1$             JSR     @APUTC
ØØØ46'ØØ24Ø1              JMP     @STC
ØØØ47'ØØØØØØ    STC:      Ø
ØØØ5Ø'ØØØ114    L:        "L
ØØØ51'ØØØ125    U:        "U
ØØØ52'ØØØ1Ø2    B:        "B
ØØØ53'ØØØØ75    EQ:       "=
ØØØ54'ØØØØ4Ø    SP:       "
```

```
ØØØØ5'ØØØØØØ    LB:       Ø
ØØØ56'ØØØØØØ    UB:       Ø
ØØØ57'ØØØØ1Ø    K:        1Ø
ØØØ6Ø'ØØØØØØ    C:        Ø
                          .END
```

```
                          .TITL   OCTBN
                          .EXTD   AGETC
                          .ENT    OCTBN,SAVE,CR
                          .NREL

ØØØØØ'Ø54414    OCTBN:    STA     3,SAVE
ØØØØ1'1524ØØ              SUB     2,2
ØØØØ2'ØØ6ØØ1$   OCTI:     JSR     @AGETC
ØØØØ3'Ø34413              LDA     3,CR
ØØØØ4'116415              SUB#    Ø,3,SNR
ØØØØ5'ØØ24Ø7              JMP     @SAVE
ØØØØ6'Ø344Ø7              LDA     3,C7
ØØØØ7'1634ØØ              AND     3,Ø
ØØØ1Ø'15312Ø              ADDZL   2,2
ØØØ11'15112Ø              MOVZL   2,2
ØØØ12'113ØØØ              ADD     Ø,2
ØØØ13'ØØØ767              JMP     OCTI
ØØØ14'ØØØØØØ    SAVE:     Ø
ØØØ15'ØØØØØ7    C7:       7
ØØØ16'ØØØØ15    CR:       15
                          .END
```

```
                              .TITL   BNOCT
                              .EXTD    APUTC
                              .ENT    BNOCT
                              .NREL

ØØØØØ'Ø54416    BNOCT:        STA     3,STORE
ØØØØ1'15262Ø                  SUBZR   2,2
ØØØØ2'Ø2Ø413    LOOP:         LDA     Ø,C6Ø
ØØØØ3'146443                  SUBO    2,1,SNC
ØØØØ4'1Ø14Ø1                  INC     Ø,Ø,SKP
ØØØØ5'147ØØ1                  ADD     2,1,SKP
ØØØØ6'ØØØ775                  JMP     .-3
ØØØØ7'ØØ6ØØ1$                 JSR     @APUTC
ØØØ1Ø'15122Ø                  MOVZR   2,2
ØØØ11'15122Ø                  MOVZR   2,2
ØØØ12'151224                  MOVZR   2,2,SZR
ØØØ13'ØØØ767                  JMP     LOOP
ØØØ14'ØØ24Ø2                  JMP     @STORE
ØØØ15'ØØØØ6Ø    C6Ø:          6Ø
ØØØ16'ØØØØØØ    STORE:        Ø
                              .END
```

```
                                .TITLE   GETC
                                .ENT     GETC,LF
                                .EXTD    APUTC,ACR
                                .NREL

ØØØØØ'Ø54417    GETC:           STA      3,SGET
ØØØØ1'Ø6Ø11Ø                    NIOS     TTI
ØØØØ2'Ø6361Ø                    SKPDN    TTI
ØØØØ3'ØØØ777                    JMP      . -1
ØØØØ4'Ø6Ø61Ø                    DIAC     Ø,TTI
ØØØØ5'Ø24413                    LDA      1,MSK
ØØØØ6'1234ØØ                    AND      1,Ø
ØØØØ7'ØØ6ØØ1$                   JSR      @APUTC
ØØØ1Ø'Ø36ØØ2$                   LDA      3,@ACR
ØØØ11'1164Ø4                    SUB      Ø,3,SZR
ØØØ12'ØØ24Ø5                    JMP      @SGET
ØØØ13'Ø2Ø4Ø6                    LDA      Ø,LF
ØØØ14'ØØ6ØØ1$                   JSR      @APUTC
ØØØ15'Ø22ØØ2$                   LDA      Ø,@ACR
ØØØ16'ØØ24Ø1                    JMP      @SGET
ØØØ17'ØØØØØØ    SGET:           Ø
ØØØ2Ø'ØØØ177    MSK:            177
ØØØ21'ØØØØ12    LF:             12
                                .END
```

```
                                .TITL   PUTC
                                .ENT    PUTC,AGETC,APUTC,AOCBN,ABNOC,ACR,ALF
                                .EXTN   GETC,OCTBN,BNOCT,CR,LF
                                .NREL

ØØØØØ'Ø63511    PUTC:           SKPBZ   TTO
ØØØØ1'ØØØ777                    JMP     .-1
ØØØØ2'Ø61111                    DOAS    Ø,TTO
ØØØØ3'1Ø1ØØ4                    MOV     Ø,Ø,SZR
ØØØØ4'ØØ14ØØ                    JMP     Ø,3
ØØØØ5'Ø544Ø7                    STA     3,SPUT
ØØØØ6'Ø22Ø45                    LDA     Ø,@ACR
ØØØØ7'ØØ4771                    JSR     PUTC
ØØØ1Ø'Ø22Ø46                    LDA     Ø,@ALF
ØØØ11'ØØ4767                    JSR     PUTC
ØØØ12'1Ø24ØØ                    SUB     Ø,Ø
ØØØ13'ØØ24Ø1                    JMP     @SPUT
ØØØ14'ØØØØØØ    SPUT:           Ø

      ØØØØ4Ø                    .LOC    4Ø
ØØØ4Ø 177777    AGETC:          GETC
ØØØ41 ØØØØØØ'   APUTC:          PUTC
ØØØ42 177777    AOCBN:          OCTBN
ØØØ43 177777    ABNOC:          BNOCT
      ØØØØ45                    .LOC    45
ØØØ45 177777    ACR:            CR
ØØØ46 177777    ALF:            LF

                                .END
```

```
                  ;LAYOUT OF STACK ENTRY

        ØØØØØØ    SAC3=Ø     ;SAVE FOR AC3
        ØØØØØ1    SACØ=1     ;SAVE FOR ACØ
        ØØØØØ2    SAC1=2     ;SAVE FOR AC1
        ØØØØØ3    SAC2=3     ;SAVE FOR AC2
        ØØØØØ4    SCRY=4     ;SAVE FOR CARRY
        ØØØØØ5    SRTN=5     ;SAVE FOR RETURN ADDRESS (WORD Ø)
        ØØØØØ6    SMSK=6     ;SAVE FOR CURRENT MASK

        ØØØØØ1               .LOC    1
ØØØØ1   ØØØ4ØØ               ISR

        ØØØ4ØØ               .LOC    4ØØ          ;LOAD IN SECOND PAGE
ØØ4ØØ   Ø56464    ISR:       STA     3,@ADSTK     ;NO-SAVE AC3 IN STACK
ØØ4Ø1   Ø34463               LDA     3,ADSTK      ;AC3 ADDRESS OF STACK
ØØ4Ø2   Ø414Ø1               STA     Ø,SACØ,3     ;SAVE ACCUMULATORS
ØØ4Ø3   Ø454Ø2               STA     1,SAC1,3
ØØ4Ø4   Ø514Ø3               STA     2,SAC2,3
ØØ4Ø5   1Ø256Ø               SUBCL   Ø,Ø          ;SAVE CARRY
ØØ4Ø6   Ø414Ø4               STA     Ø,SCRY,3
ØØ4Ø7   Ø2ØØØØ               LDA     Ø,Ø          ;SAVE RETURN ADDRESS
ØØ41Ø   Ø414Ø5               STA     Ø,SRTN,3
ØØ411   Ø2Ø456               LDA     Ø,CMASK      ;SAVE CURRENT MASK
ØØ412   Ø414Ø6               STA     Ø,SMSK,3
ØØ413   Ø3Ø455               LDA     2,SIZE       ;PUSH STACK
ØØ414   157ØØØ               ADD     2,3
ØØ415   Ø54447               STA     3,ADSTK
ØØ416   Ø61477               INTA    Ø            ;ACØ=DEVICE CODE
ØØ417   Ø3Ø446    ISR1:      LDA     2,AMTAB      ;AC2=ADDR-1 OF MASK TAB
ØØ42Ø   113ØØØ               ADD     Ø,2          ;AC2=ADDRESS OF MASK
ØØ421   Ø31ØØØ               LDA     2,Ø,2        ;AC2=NEW MASK
ØØ422   Ø5Ø445               STA     2,CMASK      ;SET CMASK TO NEW MASK
ØØ423   Ø34443               LDA     3,AJTAB      ;AC3=ADDR-1 OF JUMP TAB
ØØ424   117ØØØ               ADD     Ø,3          'AC3=ADDR OF ADDR WORD
ØØ425   Ø72177               DOBS    2,CPU        ;MSKO AND TURN ON INT
ØØ426   ØØ74ØØ               JSR     @Ø,3         ;EXIT TO ROUTINE
ØØ427   Ø6Ø277               INTDS                ;DISABLE INTERRUPTS
ØØ43Ø   Ø34434               LDA     3,ADSTK      ;POP STACK
ØØ431   Ø3Ø437               LDA     2,SIZE
ØØ432   1564ØØ               SUB     2,3
ØØ433   Ø314Ø6               LDA     2,SMSK,3     ;AC2=OLD MASK
ØØ434   Ø72Ø77               MSKO    2            ;ISSUE OLD MASK
```

```
ØØ435  Ø61477            INTA    Ø              ;GET DEVICE CODE
ØØ436  1Ø1ØØ4            MOV     Ø,Ø,SZR        ;SKIP IF NO INTS
ØØ437  ØØØ76Ø            JMP     ISR1           ;PROCESS PENDING INT
ØØ44Ø  Ø54424            STA     3,ADSTK        ;UPDATE POINTER
ØØ441  Ø5Ø426            STA     2,CMASK        ;UPDATE MASK
ØØ442  Ø214Ø5            LDA     Ø,SRTN,3       ;RESTORE RETURN ADDRESS
ØØ443  Ø4ØØØØ            STA     Ø,Ø
ØØ444  Ø214Ø4            LDA     Ø,SCRY,3       ;RESTORE CARRY
ØØ445  1Ø122Ø            MOVZR   Ø,Ø
ØØ446  Ø214Ø1            LDA     Ø,SACØ,3       ;RESTORE ACØ THRU AC2
ØØ447  Ø254Ø2            LDA     1,SAC1,3
ØØ45Ø  Ø314Ø3            LDA     2,SAC2,3
ØØ451  Ø36413            LDA     3,@ADSTK       ;RESTORE AC3
ØØ452  Ø6Ø177            INTEN                  ;ENABLE INTERRUPTS
ØØ453  ØØ2ØØØ            JMP     @Ø             ;RETURN TO ROUTINE

                 ;ROUTINE TO IGNORE INTERRUPTS.

ØØ454  Ø244Ø5  IGNOR:    LDA     1,CLEAR        ;LOAD NIOC COMMAND
ØØ455  123ØØØ            ADD     1,Ø            ;ADD IN DEVICE CODE
ØØ456  Ø4Ø4Ø1            STA     Ø,.+1          ;STORE IN NEXT
ØØ457  ØØØØØØ            Ø                      ;EXECUTE NIOC COMMAND
ØØ46Ø  ØØ14ØØ            JMP     Ø,3            ;RETURN TO ROUTINE
ØØ461  Ø6Ø2ØØ  CLEAR:    NIOC    Ø

                 ;ERROR HALTS.

ØØ462  Ø63Ø77  ERROR:    HALT
ØØ463  ØØØ771            JMP     IGNOR

                 ;STORAGE AND ADDRESS CONSTANTS.

ØØ464  ØØØ545  ADSTK:    STACK                  ;ADDRESS OF PUSHDOWN STACK
ØØ465  ØØØ474  AMTAB:    MTAB-1                 ;ADDR-1 OF MASK TABLE
ØØ466  ØØØ5Ø6  AJTAB:    JTAB-1                 ;ADDR-1 OF JUMP TABLE
ØØ467  ØØØØØØ  CMASK:    Ø                      ;STORAGE FOR CURRENT MASK
ØØ47Ø  ØØØØØ7  SIZE:     7                      ;SIZE OF STACK ENTRY (7 WORDS)
```

```
                    ;MASK TABLE.

          177777    ALL=177777            ;MASK TO DISABLE ALL INTERRUPTS.
ØØ471     177777    MTAB:     ALL
ØØ472     177777              ALL
ØØ473     177777              ALL
ØØ474     177777              ALL
ØØ475     177777              ALL
ØØ476     177777              ALL
ØØ477     177777              ALL
ØØ5ØØ     177777              ALL
ØØ5Ø1     177777              ALL
ØØ5Ø2     177777              ALL

                    ;JUMP TABLE.

          ØØØ464    ERR=ERROR

ØØ5Ø3     ØØØ464    JTAB:     ERR
ØØ5Ø4     ØØØ464              ERR
ØØ5Ø5     ØØØ464              ERR
ØØ5Ø6     ØØØ464              ERR
ØØ5Ø7     ØØØ464              ERR
ØØ51Ø     ØØØ464              ERR
ØØ511     ØØØ464              ERR
ØØ512     ØØØ464              ERR
ØØ513     ØØØ464              ERR
ØØ514     ØØØ464              ERR

                    ;INITIALIZATION ROUTINE.

ØØ515     Ø2442Ø    INIT:     LDA    1,ASTK       ;INITIALIZE POINTER
ØØ516     Ø44746              STA    1,ADSTK
ØØ517     1264ØØ              SUB    1,1          ;ZERO CURRENT MASK
ØØ52Ø     Ø44747              STA    1,CMASK
ØØ521     Ø2Ø415              LDA    Ø,ADERR      ;ACØ=A (ERROR ROUTINE)
ØØ522     Ø24415              LDA    1,MALL       ;AC1=FULL MASK
ØØ523     Ø3Ø415              LDA    2,M12        ;AC2=-1Ø
ØØ524     Ø34741              LDA    3,AMTAB      ;MEM(2Ø)=A(MTAB)-1
ØØ525     Ø54Ø2Ø              STA    3,2Ø
ØØ526     Ø3474Ø              LDA    3,AJTAB      ;MEM(21)=A(JTAB)-1
ØØ527     Ø54Ø21              STA    3,21
ØØ53Ø     Ø42Ø21    INIT1:    STA    Ø,@21        ;ENTER IN JTAB
ØØ531     Ø46Ø2Ø              STA    1,@2Ø        ;ENTER IN MTAB
ØØ532     1514Ø4              INC    2,2,SZR      ;LOOP 1Ø  TIMES
ØØ533     ØØØ775              JMP    INIT1
ØØ534     Ø63Ø77              HALT
```

```
ØØ535 ØØØ545    ASTK:     STACK  ;ADDRESS OF STACK
ØØ536 ØØØ464    ADERR:    ERROR  ;ADDRESS OF ERROR ROUTINE
ØØ537 177777    MALL:     ALL    ;MASK TO ENABLE ALL INTS
ØØ540 177766    M12:      -12    ;MINUS 1Ø

      ØØØØ43    STACK:    .BLK   5*7

                          .END
```

APPENDIX B

BIBLIOGRAPHY

More complete information on the programs mentioned in this manual is contained in the documents listed here.

ASSEMBLERS

Absolute

Document: 093-000017

The Absolute Assembler is a two-pass assembler accepting symbolic input and producing absolute binary output or an assembly listing or both. Pseudo commands are available to alter the program origin, change the current number radix, and define new operation codes. Source input is free-form, using special characters to delimit labels and comments. Assembly speed is entirely I/O limited. The assembler is approximately 5000 (octal) words in length and uses all remaining memory locations for symbol table storage.

Extended

Document: 093-000040

The Extended Assembler, like the Absolute Assembler, converts symbolic assembly statements into machine language code. In addition to Absolute Assembler features, the Extended Assembler provides relocation, interprogram communication, conditional assembly, and more powerful number definition facilities. It contains about 7,400 (octal) instructions and uses the remainder of memory for symbol table storage.

DEBUGGERS

Debug I

Document: 093-000038

Debug I is a software debugging routine that allows one breakpoint. Virtually no restrictions are applied to its placement or use. Debug I can interface with any type of routine, including those using the Nova interrupt hardware. Debug I requires only 300 (octal) locations.

Debug II

Document: 093-000020

Debug II is a software routine that allows for simultaneous activation of up to four breakpoints. Virtually no restrictions are applied to their placement or use. Debug II can interface with any type of routine, including those using the interrupt hardware. Commands are provided for examining, searching, and altering memory, as well as punching ranges of memory in absolute binary format. This program consists of less than 1400 (octal) instructions.

Debug III — Document: 093-000044

Debug III is a routine for symbolic debugging of user programs. It provides all of the features of Debug II in addition to those following. Instructions may be input in symbolic format in a manner similar to the symbolic input to the assembler. Further, symbols defined at assembly time can be output as part of the user's relocatable binary, loaded by the relocatable loader, and accessed by Debug III. This provides great flexibility for symbolic debugging at run time, giving the user access to all symbolic information known at assembly and load time. Further, eight breakpoints may be active at one time. Debug III requires approximately 4000 (octal) locations and is supplied in relocatable binary format.

EDITING ROUTINES

Editor — Document: 093-000018

The Editor is a routine that enables editing of source input to produce updated source output. It is most commonly used to modify program source tapes in preparation for a new assembly. The Editor executes simple command strings, input using the teletype, to modify text on either a character or a line basis. The location of specific text is facilitated by means of string searches. The program is less than 2000 (octal) words in length.

Macro Editor — Document: 093-000018

The Macro Editor provides all the features of the Editor and in addition allows the user to define command strings in a special "macro" register. The command string can then be executed repeatedly by merely specifying the macro register name in further command strings.

FLOATING POINT INTERPRETER — Document: 093-000019

The Floating Point Interpreter is a program designed to expand the instruction set to include over thirty-five additional instructions. These instructions cover a wide range of floating point operations, floating point conversions, and transcendental function operations. Numbers are represented in floating hexadecimal, providing the user with 7 significant digits and an approximate range in magnitude of 10**-78 to 10**+75. The Basic Floating Point Interpreter is 2000 (octal) locations in length. The Extended version is approximately 3500 (octal) locations in length. The interpreter is supplied in both absolute and relocatable binary formats. The absolute version is origined to occupy the upper locations of a 4K memory.

LOADERS

Bootstrap

Document: 093-000002

The Bootstrap Loader is a short routine to load the Binary Loader into memory. The Bootstrap requires 15 (octal) words and 2 temporary locations.

Supernova Selfload Bootstrap

Document: 093-000055

The Selfload tape is used in conjunction with the program load feature of the Supernova to place an Absolute Binary Loader in the highest locations of alterable storage. This **program** contains 40 (octal) instructions.

Nova 800/1200 Selfload Bootstrap

Document: 093-000055

This program is used in conjunction with the optional program load feature of the Nova 1200 or Nova 800. This feature automatically loads the Absolute Binary Loader into the highest locations of the machine's alterable storage, using a bootstrap program implemented in hardware.

Absolute Binary Loader

Document: 093-000003

The Absolute Binary Loader is a routine used to load any absolute binary tapes, such as those produced as output by the Absolute Assembler. The Loader is 120 (octal) words in length, 116 of which immediately precede the Bootstrap Loader in memory. The speed of the Binary Loader is limited by the speed of the input device.

Relocatable Binary Loader

Document: 093-000039

This program is used to load relocatable binary tapes produced as output by the Extended Assembler. The loader accepts any number of relocatable binary tapes as input, resolves external displacements and normal externals, and maintains an entry symbol table that can be printed on demand. This routine consists of less than 2200 (octal) instructions.

SYSTEM REFERENCE MANUAL

"How to Use the Nova Computers"

The system reference manual:

1. Complements the material contained in the Assembler manuals.
2. Contains more advanced and detailed information on programming of the Nova Computers than is found in this manual, "Introduction to Programming the Nova Computers".
3. Supplies needed information on equipment available, interfacing and installation.

APPENDIX C

ASSEMBLER PSEUDO-OPS

.BLK expression	Allocate a storage block by incrementing the location counter by expression.
.DALC equivalence statement	Define a symbol for an arithmetic and logical instruction.
.DIAC equivalence statement	Define a symbol for an instruction that will replace bits 3 and 4 with an AC number.
.DIO equivalence statement	Define a symbol for an I/O instruction having only a device code.
.DIOA equivalence statement	Define a symbol for an I/O instruction having a device code and accumulator.
.DMR equivalence statement	Define a symbol for a memory reference instruction that does not use an accumulator.
.DMRA equivalence statement	Define a symbol for a memory reference instruction that does use an accumulator.
.DUSR equivalence statement	Define a user symbol.
.END [expression]	Terminate source program. The optional expression evaluates to a location to which to transfer when the object tape is loaded.
.ENDC	Terminate conditional assembly coding.
.ENT $symbol_1$, [$symbol_2$...]	Define an entry within a program that can be referenced by another program in which the symbol has been declared .EXTN or .EXTD (normal external or displacement external.)
.EOT	End of tape, implying there is another source program tape. Assembler halts, allowing the operator to mount the next tape and to press CONTINUE to continue assembly.

.EXTD symbol$_1$, [symbol$_2$ ...]	Declare a symbol as a displacement external.
.EXTN symbol$_1$, [symbol$_2$ ...]	Declare a symbol as a normal external.
.IFE expression	Assemble statements following until a .ENDC is encountered if expression evaluates to zero in pass 1.
.IFN expression	Assemble statements following until a .ENDC is encountered if expression does not evaluate to zero in pass 1.
.LOC expression	Set the location counter to the value of expression.
.NREL	Assemble instructions following as normal relocatable.
.RDX expression	Interpret integers following as having the radix given by expression.
.TITL name	Define a symbol as the name of the program.
.TXT *message*	Store characters of message two per word (one per 8-bit byte). * represents any delimiting symbol not in the text. The rightmost 7 bits are used to store the character and the leftmost bit is determined as follows:

	Value of Left Bit
.TXT *message*	- zero
.TXTE *message*	- even parity for byte
.TXTF *message*	- one
.TXTO *message*	- odd parity for byte

.TXTM expression	Change packing mode of text. Default packing is right to left. If a .TXTM is encountered and expression evaluates to non-zero, packing is left to right; if it evaluates to zero packing is right to left.
.XPNG	Undefine all previously defined (.DUSR, etc.) symbols except permanent symbols.
.ZREL	Assemble statements following as zero relocatable.

APPENDIX D

INSTRUCTION MNEMONICS AND TIMING

The table beginning on the next page gives the instruction menemonics in numerical order. Following that is an alphabetic listing that gives the octal value and a short description of the instruction. Instruction execution times in microseconds are listed on page D-12.

The derivations of the instruction mnemonics are as follows:

$$\left.\begin{array}{l}\text{LoaD}\\\text{STore}\end{array}\right\}\ \text{Accumulator}$$

$$\left.\begin{array}{l}\text{Increment}\\\text{Decrement}\end{array}\right\}\ \text{and Skip if Zero}$$

JuMP

Jump to SubRoutine

$$\left.\begin{array}{l}\text{COMplement}\\\text{NEGate}\\\text{MOVe}\\\text{INCrement}\\\text{ADd Complement}\\\text{SUBtract}\\\text{ADD}\\\text{AND}\end{array}\right\}\ \begin{array}{l}\text{for carry bit}\\\text{base value use}\end{array}\ \left\{\begin{array}{l}\text{current carry}\\\text{Zero}\\\text{One}\\\text{Complement of current carry}\end{array}\right\}\ \left\{\begin{array}{l}\sim\\\text{shift Left}\\\text{shift Right}\\\text{Swap bytes}\end{array}\right\}\ \left\{\begin{array}{l}\sim\\\#\end{array}\right.$$

$$\begin{array}{l}\text{SKiP}\\\text{Skip}\end{array}\ \left\{\begin{array}{l}\left.\begin{array}{l}\text{on Zero}\\\text{on Nonzero}\end{array}\right\}\ \left\{\begin{array}{l}\text{Carry}\\\text{Result}\end{array}\right.\\\text{if Either is Zero}\\\text{if Both are Nonzero}\end{array}\right.$$

$$\left.\begin{array}{l}\text{No IO transfer}\\\text{Data}\left\{\begin{array}{l}\text{In}\\\text{Out}\end{array}\right\}\left\{\begin{array}{l}\text{A}\\\text{B}\\\text{C}\end{array}\right\}\text{buffer}\end{array}\right]\ \text{and}\ \left\{\begin{array}{l}\sim\\\text{Start}\\\text{Clear}\\\text{special Pulse}\end{array}\right.$$

$$\text{SKiP if}\left\{\begin{array}{l}\text{Busy}\\\text{Done}\end{array}\right\}\text{is}\left\{\begin{array}{l}\text{Nonzero}\\\text{Zero}\end{array}\right.$$

READ Switches

IO ReSeT

HALT

INTerrupt Acknowledge

MaSK Out

INTerrupt ENable

INTerrupt DiSable

MULtiply

DIVide

INSTRUCTION MNEMONICS

NUMERIC LISTING

000000	JMP	062677	IORST	100350	COMOS#
000001	SKP	062700	DICP	100360	COMCS
000002	SZC	063000	DOC	100370	COMCS#
000003	SNC	063077	HALT	100400	NEG
000004	SZR	063100	DOCS	100410	NEG#
000005	SNR	063200	DOCC	100420	NEGZ
000006	SEZ	063300	DOCP	100430	NEGZ#
000007	SBN	063400	SKPBN	100440	NEGO
000010	#	063500	SKPBZ	100450	NEGO#
002000	@	063600	SKPDN	100460	NEGC
004000	JSR	063700	SKPDZ	100470	NEGC#
010000	ISZ	073101	DIV	100500	NEGL
014000	DSZ	073301	MUL	100510	NEGL#
020000	LDA	100000	@	100520	NEGZL
040000	STA	100000	COM	100530	NEGZL#
060000	NIO	100010	COM#	100540	NEGOL
060100	NIOS	100020	COMZ	100550	NEGOL#
060177	INTEN	100030	COMZ#	100560	NEGCL
060200	NIOC	100040	COMO	100570	NEGCL#
060277	INTDS	100050	COMO#	100600	NEGR
060300	NIOP	100060	COMC	100610	NEGR#
060400	DIA	100070	COMC#	100620	NEGZR
060477	READS	100100	COML	100630	NEGZR#
060500	DIAS	100110	COML#	100640	NEGOR
060600	DIAC	100120	COMZL	100650	NEGOR#
060700	DIAP	100130	COMZL#	100660	NEGCR
061000	DOA	100140	COMOL	100670	NEGCR#
061100	DOAS	100150	COMOL#	100700	NEGS
061200	DOAC	100160	COMCL	100710	NEGS#
061300	DOAP	100170	COMCL#	100720	NEGZS
061400	DIB	100200	COMR	100730	NEGZS#
061477	INTA	100210	COMR#	100740	NEGOS
061500	DIBS	100220	COMZR	100750	NEGOS#
061600	DIBC	100230	COMZR#	100760	NEGCS
061700	DIBP	100240	COMOR	100770	NEGCS#
062000	DOB	100250	COMOR#	101000	MOV
062077	MSKO	100260	COMCR	101010	MOV#
062100	DOBS	100270	COMCR#	101020	MOVZ
062200	DOBC	100300	COMS	101030	MOVZ#
062300	DOBP	100310	COMS#	101040	MOVO
062400	DIC	100320	COMZS	101050	MOVO#
062500	DICS	100330	COMZS#	101060	MOVC
062600	DICC	100340	COMOS	101070	MOVC#

101100	MOVL	101660	INCCR	102440	SUBO
101110	MOVL#	101670	INCCR#	102450	SUBO#
101120	MOVZL	101700	INCS	102460	SUBC
101130	MOVZL#	101710	INCS#	102470	SUBC#
101140	MOVOL	101720	INCZS	102500	SUBL
101150	MOVOL#	101730	INCZS#	102510	SUBL#
101160	MOVCL	101740	INCOS	102520	SUBZL
101170	MOVCL#	101750	INCOS#	102530	SUBZL#
101200	MOVR	101760	INCCS	102540	SUBOL
101210	MOVR#	101770	INCCS#	102550	SUBOL#
101220	MOVZR	102000	ADC	102560	SUBCL
101230	MOVZR#	102010	ADC#	102570	SUBCL#
101240	MOVOR	102020	ADCZ	102600	SUBR
101250	MOVOR#	102030	ADCZ#	102610	SUBR#
101260	MOVCR	102040	ADCO	102620	SUBZR
101270	MOVCR#	102050	ADCO#	102630	SUBZR#
101300	MOVS	102060	ADCC	102640	SUBOR
101310	MOVS#	102070	ADCC#	102650	SUBOR#
101320	MOVZS	102100	ADCL	102660	SUBCR
101330	MOVZS#	102110	ADCL#	102670	SUBCR#
101340	MOVOS	102120	ADCZL	102700	SUBS
101350	MOVOS#	102130	ADCZL#	102710	SUBS#
101360	MOVCS	102140	ADCOL	102720	SUBZS
101370	MOVCS#	102150	ADCOL#	102730	SUBZS#
101400	INC	102160	ADCCL	102740	SUBOS
101410	INC#	102170	ADCCL#	102750	SUBOS#
101420	INCZ	102200	ADCR	102760	SUBCS
101430	INCZ#	102210	ADCR#	102770	SUBCS#
101440	INCO	102220	ADCZR	103000	ADD
101450	INCO#	102230	ADCZR#	103010	ADD#
101460	INCC	102240	ADCOR	103020	ADDZ
101470	INCC#	102250	ADCOR#	103030	ADDZ#
101500	INCL	102260	ADCCR	103040	ADDO
101510	INCL#	102270	ADCCR#	103050	ADDO#
101520	INCZL	102300	ADCS	103060	ADDC
101530	INCZL#	102310	ADCS#	103070	ADDC#
101540	INCOL	102320	ADCZS	103100	ADDL
101550	INCOL#	102330	ADCZS#	103110	ADDL#
101560	INCCL	102340	ADCOS	103120	ADDZL
101570	INCCL#	102350	ADCOS#	103130	ADDZL#
101600	INCR	102360	ADCCS	103140	ADDOL
101610	INCR#	102370	ADCCS#	103150	ADDOL#
101620	INCZR	102400	SUB	103160	ADDCL
101630	INCZR#	102410	SUB#	103170	ADDCL#
101640	INCOR	102420	SUBZ	103200	ADDR
101650	INCOR#	102430	SUBZ#	103210	ADDR#

103220	ADDZR
103230	ADDZR#
103240	ADDOR
103250	ADDOR#
103260	ADDCR
103270	ADDCR#
103300	ADDS
103310	ADDS#
103320	ADDZS
103330	ADDZS#
103340	ADDOS
103350	ADDOS#
103360	ADDCS
103370	ADDCS#
103400	AND
103410	AND#
103420	ANDZ
103430	ANDZ#
103440	ANDO
103450	ANDO#
103460	ANDC
103470	ANDC#
103500	ANDL
103510	ANDL#
103520	ANDZL
103530	ANDZL#
103540	ANDOL
103550	ANDOL#
103560	ANDCL
103570	ANDCL#
103600	ANDR
103610	ANDR#
103620	ANDZR
103630	ANDZR#
103640	ANDOR
103650	ANDOR#
103660	ANDCR
103670	ANDCR#
103700	ANDS
103710	ANDS#
103720	ANDZS
103730	ANDZS#
103740	ANDOS
103750	ANDOS#
103760	ANDCS
103770	ANDCS#

INSTRUCTION MNEMONICS
ALPHABETIC LISTING

ADC	102000	Add the complement of ACS to ACD; use Carry as base for carry bit.
ADCC	102060	Add the complement of ACS to ACD; use complement of Carry as base for carry bit.
ADCCL	102160	Add the complement of ACS to ACD; use complement of Carry as base for carry bit; rotate left.
ADCCR	102260	Add the complement of ACS to ACD; use complement of Carry as base for carry bit; rotate right.
ADCCS	102360	Add the complement of ACS to ACD; use complement of Carry as base for carry bit; swap halves of result.
ADCL	102100	Add the complement of ACS to ACD; use Carry as base for carry bit; rotate left.
ADCO	102040	Add the complement of ACS to ACD; use 1 as base for carry bit.
ADCOL	102140	Add the complement of ACS to ACD; use 1 as base for carry bit; rotate left.
ADCOR	102240	Add the complement of ACS to ACD; use 1 as base for carry bit; rotate right.
ADCOS	102340	Add the complement of ACS to ACD; use 1 as base for carry bit; swap halves of result.
ADCR	102200	Add the complement of ACS to ACD; use Carry as base for carry bit; rotate right.
ADCS	102300	Add the complement of ACS to ACD; use Carry as base for carry bit; swap halves of result.
ADCZ	102020	Add the complement of ACS to ACD; use 0 as base for carry bit.
ADCZL	102120	Add the complement of ACS to ACD; use 0 as base for carry bit; rotate left.
ADCZR	102220	Add the complement of ACS to ACD; use 0 as base for carry bit; rotate right.
ADCZS	102320	Add the complement of ACS to ACD; use 0 as base for carry bit; swap halves of result.
ADD	103000	Add ACS to ACD; use Carry as base for carry bit.
ADDC	103060	Add ACS to ACD; use complement of Carry as base for carry bit.
ADDCL	103160	Add ACS to ACD; use complement of Carry as base for carry bit; rotate left.
ADDCR	103260	Add ACS to ACD; use complement of Carry as base for carry bit; rotate right.
ADDCS	103360	Add ACS to ACD; use complement of Carry as base for carry bit; swap halves of result.
ADDL	103100	Add ACS to ACD; use Carry as base for carry bit; rotate left.
ADDO	103040	Add ACS to ACD; use 1 as base for carry bit.
ADDOL	103140	Add ACS to ACD; use 1 as base for carry bit; rotate left.

ADDOR	103240	Add ACS to ACD; use 1 as base for carry bit; rotate right.
ADDOS	103340	Add ACS to ACD; use 1 as base for carry bit; swap halves of result.
ADDR	103200	Add ACS to ACD; use Carry as base for carry bit; rotate right.
ADDS	103300	Add ACS to ACD; use Carry as base for carry bit; swap halves of result.
ADDZ	103020	Add ACS to ACD; use 0 as base for carry bit.
ADDZL	103120	Add ACS to ACD; use 0 as base for carry bit; rotate left.
ADDZR	103220	Add ACS to ACD; use 0 as base for carry bit; rotate right.
ADDZS	103320	Add ACS to ACD; use 0 as base for carry bit; swap halves of result.
AND	103400	And ACS with ACD; use Carry as carry bit.
ANDC	103460	And ACS with ACD; use complement of Carry as carry bit.
ANDCL	103560	And ACS with ACD; use complement of Carry as carry bit; rotate left.
ANDCR	103660	And ACS with ACD; use complement of Carry as carry bit; rotate right.
ANDCS	103760	And ACS with ACD; use complement of Carry as carry bit; swap halves of result.
ANDL	103500	And ACS with ACD; use Carry as carry bit; rotate left.
ANDO	103440	And ACS with ACD; use 1 as carry bit.
ANDOL	103540	And ACS with ACD; use 1 as carry bit; rotate left.
ANDOR	103640	And ACS with ACD; use 1 as carry bit; rotate right.
ANDOS	103740	And ACS with ACD; use 1 as carry bit; swap halves of result.
ANDR	103600	And ACS with ACD; use Carry as carry bit; rotate right.
ANDS	103700	And ACS with ACD; use Carry as carry bit; swap halves of result.
ANDZ	103420	And ACS with ACD; use 0 as carry bit.
ANDZL	103520	And ACS with ACD; use 0 as carry bit; rotate left.
ANDZR	103620	And ACS with ACD; use 0 as carry bit; rotate right.
ANDZS	103720	And ACS with ACD; use 0 as carry bit; swap halves of result.
COM	100000	Place the complement of ACS in ACD; use Carry as carry bit.
COMC	100060	Place the complement of ACS in ACD; use complement of Carry as carry bit.
COMCL	100160	Place the complement of ACS in ACD; use complement of Carry as carry bit; rotate left.
COMCR	100260	Place the complement of ACS in ACD; use complement of Carry as carry bit; rotate right.
COMCS	100360	Place the complement of ACS in ACD; use complement of Carry as carry bit; swap halves of result.
COML	100100	Place the complement of ACS in ACD; use Carry as carry bit; rotate left.
COMO	100040	Place the complement of ACS in ACD; use 1 as carry bit.
COMOL	100140	Place the complement of ACS in ACD; use 1 as carry bit; rotate left.

COMOR	100240	Place the complement of ACS in ACD; use 1 as carry bit; rotate right.
COMOS	100340	Place the complement of ACS in ACD; use 1 as carry bit; swap halves of result.
COMR	100200	Place the complement of ACS in ACD; use Carry as carry bit; rotate right.
COMS	100300	Place the complement of ACS in ACD; use Carry as carry bit; swap halves of result.
COMZ	100020	Place the complement of ACS in ACD; use 0 as carry bit.
COMZL	100120	Place the complement of ACS in ACD; use 0 as carry bit; rotate left.
COMZR	100220	Place the complement of ACS in ACD; use 0 as carry bit; rotate right.
COMZS	100320	Place the complement of ACS in ACD; use 0 as carry bit; swap halves of result.
DIA	060400	Data in, A buffer to AC.
DIAC	060600	Data in, A buffer to AC; clear device.
DIAP	060700	Data in, A buffer to AC; send special pulse to device.
DIAS	060500	Data in, A buffer to AC; start device.
DIB	061400	Data in, B buffer to AC.
DIBC	061600	Data in, B buffer to AC; clear device.
DIBP	061700	Data in, B buffer to AC; send special pulse to device.
DIBS	061500	Data in, B buffer to AC; start device.
DIC	062400	Data in, C buffer to AC.
DICC	062600	Data in, C buffer to AC; clear device.
DICP	062700	Data in, C buffer to AC; send special pulse to device.
DICS	062500	Data in, C buffer to AC; start device.
DIV	073101	If overflow, set Carry. Otherwise divide AC0-AC1 by AC2. Put quotient in AC1, remainder in AC0.
DOA	061000	Data out, AC to A buffer.
DOAC	061200	Data out, AC to A buffer; clear device.
DOAP	061300	Data out, AC to A buffer; send special pulse to device.
DOAS	061100	Data out, AC to A buffer; start device.
DOB	062000	Data out, AC to B buffer.
DOBC	062200	Data out, AC to B buffer; clear device.
DOBP	062300	Data out, AC to B buffer; send special pulse to device.
DOBS	062100	Data out, AC to B buffer; start device.
DOC	063000	Data out, AC to C buffer.
DOCC	063200	Data out, AC to C buffer; clear device.
DOCP	063300	Data out, AC to C buffer; send special pulse to device.
DOCS	063100	Data out, AC to C buffer; start device.
DSZ	014000	Decrement location *E* by 1 and skip if result is zero.

HALT	063077	Halt the processor (= DOC 0,CPU).
INC	101400	Place ACS + 1 in ACD; use Carry as base for carry bit.
INCC	101460	Place ACS + 1 in ACD; use complement of Carry as base for carry bit.
INCCL	101560	Place ACS + 1 in ACD; use complement of Carry as base for carry bit; rotate left.
INCCR	101660	Place ACS + 1 in ACD; use complement of Carry as base for carry bit; rotate right.
INCCS	101760	Place ACS + 1 in ACD; use complement of Carry as base for carry bit; swap halves of result.
INCL	101500	Place ACS + 1 in ACD; use Carry as base for carry bit; rotate left.
INCO	101440	Place ACS + 1 in ACD; use 1 as base for carry bit.
INCOL	101540	Place ACS + 1 in ACD; use 1 as base for carry bit; rotate left.
INCOR	101640	Place ACS + 1 in ACD; use 1 as base for carry bit; rotate right.
INCOS	101740	Place ACS + 1 in ACD; use 1 as base for carry bit; swap halves of result.
INCR	101600	Place ACS + 1 in ACD; use Carry as base for carry bit; rotate right.
INCS	101700	Place ACS + 1 in ACD; use Carry as base for carry bit; swap halves of result.
INCZ	101420	Place ACS + 1 in ACD; use 0 as base for carry bit.
INCZL	101520	Place ACS + 1 in ACD; use 0 as base for carry bit; rotate left.
INCZR	101620	Place ACS + 1 in ACD; use 0 as base for carry bit; rotate right.
INCZS	101720	Place ACS + 1 in ACD; use 0 as base for carry bit; swap halves of result.
INTA	061477	Acknowledge interrupt by loading code of nearest device that is requesting an interrupt into AC bits 10–15 (= DIB –,CPU).
INTDS	060277	Disable interrupt by clearing Interrupt On (= NIOC CPU).
INTEN	060177	Enable interrupt by setting Interrupt On (= NIOS CPU).
IORST	062677	Clear all IO devices, clear Interrupt On, reset clock to line frequency (= DICC 0,CPU).
ISZ	010000	Increment location *E* by 1 and skip if result is zero.
JMP	000000	Jump to location *E* (put *E* in PC).
JSR	004000	Load PC + 1 in AC3 and jump to subroutine at location *E* (put *E* in PC).
LDA	020000	Load contents of location *E* into AC.
MOV	101000	Move ACS to ACD; use Carry as carry bit.
MOVC	101060	Move ACS to ACD; use complement of Carry as carry bit.
MOVCL	101160	Move ACS to ACD; use complement of Carry as carry bit; rotate left.
MOVCR	101260	Move ACS to ACD; use complement of Carry as carry bit; rotate right.
MOVCS	101360	Move ACS to ACD; use complement of Carry as carry bit; swap halves of result.
MOVL	101100	Move ACS to ACD; use Carry as carry bit; rotate left.

MOVO	101040	Move ACS to ACD; use 1 as carry bit.
MOVOL	101140	Move ACS to ACD; use 1 as carry bit; rotate left.
MOVOR	101240	Move ACS to ACD; use 1 as carry bit; rotate right.
MOVOS	101340	Move ACS to ACD; use 1 as carry bit; swap halves of result.
MOVR	101200	Move ACS to ACD; use Carry as carry bit; rotate right.
MOVS	101300	Move ACS to ACD; use Carry as carry bit; swap halves of result.
MOVZ	101020	Move ACS to ACD; use 0 as carry bit.
MOVZL	101120	Move ACS to ACD; use 0 as carry bit; rotate left.
MOVZR	101220	Move ACS to ACD; use 0 as carry bit; rotate right.
MOVZS	101320	Move ACS to ACD; use 0 as carry bit; swap halves of result.
MSKO	062077	Set up Interrupt Disable flags according to mask in AC (= DOB -,CPU).
MUL	073301	Multiply AC1 by AC2, add product to AC0, put result in AC0-AC1.
NEG	100400	Place negative of ACS in ACD; use Carry as base for carry bit.
NEGC	100460	Place negative of ACS in ACD; use complement of Carry as base for carry bit.
NEGCL	100560	Place negative of ACS in ACD; use complement of Carry as base for carry bit; rotate left.
NEGCR	100660	Place negative of ACS in ACD; use complement of Carry as base for carry bit; rotate right.
NEGCS	100760	Place negative of ACS in ACD; use complement of Carry as base for carry bit; swap halves of result.
NEGL	100500	Place negative of ACS in ACD; use Carry as base for carry bit; rotate left.
NEGO	100440	Place negative of ACS in ACD; use 1 as base for carry bit.
NEGOL	100540	Place negative of ACS in ACD; use 1 as base for carry bit; rotate left.
NEGOR	100640	Place negative of ACS in ACD; use 1 as base for carry bit; rotate right.
NEGOS	100740	Place negative of ACS in ACD; use 1 as base for carry bit; swap halves of result.
NEGR	100600	Place negative of ACS in ACD; use Carry as carry bit; rotate right.
NEGS	100700	Place negative of ACS in ACD; use Carry as carry bit; swap halves of result.
NEGZ	100420	Place negative of ACS in ACD; use 0 as base for carry bit.
NEGZL	100520	Place negative of ACS in ACD; use 0 as base for carry bit; rotate left.
NEGZR	100620	Place negative of ACS in ACD; use 0 as base for carry bit; rotate right.
NEGZS	100720	Place negative of ACS in ACD; use 0 as base for carry bit; swap halves of result.
NIO	060000	No operation.
NIOC	060200	Clear device.
NIOP	060300	Send special pulse to device.

NIOS	060100	Start device.
READS	060477	Read console data switches into AC (= DIA –,CPU).
SBN	000007	Skip if both carry and result are nonzero (skip function in an arithmetic or logical instruction).
SEZ	000006	Skip if either carry or result is zero (skip function in an arithmetic or logical instruction).
SKP	000001	Skip (skip function in an arithmetic or logical instruction).
SKPBN	063400	Skip if Busy is 1.
SKPBZ	063500	Skip if Busy is 0.
SKPDN	063600	Skip if Done is 1.
SKPDZ	063700	Skip if Done is 0.
SNC	000003	Skip if carry bit is 1 (skip function in an arithmetic or logical instruction).
SNR	000005	Skip if result is nonzero (skip function in an arithmetic or logical instruction).
STA	040000	Store AC in location *E*.
SUB	102400	Subtract ACS from ACD; use Carry as base for carry bit.
SUBC	102460	Subtract ACS from ACD; use complement of Carry as base for carry bit.
SUBCL	102560	Subtract ACS from ACD; use complement of Carry as base for carry bit; rotate left.
SUBCR	102660	Subtract ACS from ADC; use complement of Carry as base for carry bit; rotate right.
SUBCS	102760	Subtract ACS from ACD; use complement of Carry as base for carry bit; swap halves of result.
SUBL	102500	Subtract ACS from ACD; use Carry as base for carry bit; rotate left.
SUBO	102440	Subtract ACS from ACD; use 1 as base for carry bit.
SUBOL	102540	Subtract ACS from ACD; use 1 as base for carry bit; rotate left.
SUBOR	102640	Subtract ACS from ACD; use 1 as base for carry bit; rotate right.
SUBOS	102740	Subtract ACS from ACD; use 1 as base for carry bit; swap halves of result.
SUBR	102600	Subtract ACS from ACD; use Carry as base for carry bit; rotate right.
SUBS	102700	Subtract ACS from ACD; use Carry as base for carry bit; swap halves of result.
SUBZ	102420	Subtract ACS from ACS; use 0 as base for carry bit.
SUBZL	102520	Subtract ACS from ACD; use 0 as base for carry bit; rotate left.
SUBZR	102620	Subtract ACS from ACD; use 0 as base for carry bit; rotate right.
SUBZS	102720	Subtract ACS from ACD; use 0 as base for carry bit; swap halves of result.
SZC	000002	Skip if carry is 0 (skip function in an arithmetic or logical instruction).
SZR	000004	Skip if result is zero (skip function in an arithmetic or logical instruction).

@	002000	When this character appears in an instruction, the assembler places a 1 in bit 5 to produce indirect addressing.
@	100000	When this character appears with a 15-bit address, the assembler places a 1 in bit 0, making the address indirect.
#	000010	Appending this character to the mnemonic for an arithmetic or logical instruction places a 1 in bit 12 to prevent the processor from loading the 17-bit result in Carry and ACD. Thus the result of an instruction can be tested for a skip without affecting Carry or the accumulators.

INSTRUCTION EXECUTION TIMES

Supernova read-only time equals semiconductor time, except add .2 for LDA, STA, ISZ, DSZ if reference is to core. Nova times are for core; for read-only subtract .2 except subtract .4 for LDA, STA, ISZ, DSZ if reference is to read-only memory.

When two numbers are given, the one at the left of the slash is the time for an isolated transfer, the one at the right is the minimum time between consecutive transfers.

	Supernova SC	Supernova Core	Nova 800	Nova 1200	Nova
LDA	1.2	1.6	1.6	2.55	5.2
STA	1.2	1.6	1.6	2.55	5.5
ISZ, DSZ	1.4	1.8	1.8	3.15*	5.2
JMP	.6	.8	.8	1.35	2.6
JSR	1.2	1.4	.8	1.35	3.5
Indirect addressing add	.6	.8	.8	1.2	2.6
Base register addressing add	0	0	0	0	.3
Autoindexing add	.2	.2	.2	.6	0
COM, NEG, MOV, INC	.3*	.8*	.8*	1.35*	5.6
ADC, SUB, ADD, AND	.3*	.8*	.8*	1.35*	5.9
*If skip occurs add	‡	.8	.2	1.35	
IO input (except INTA)	2.8	2.9	2.2†	2.55	4.4
NIO	3.2	3.3	2.2†	3.15	4.4
IO output	3.2	3.3	2.2†	3.15	4.7
†S, C or P add			.6		
IO skips	2.8	2.9	1.4*	2.55	4.4
INTA	3.6	3.7	2.2	2.55	4.4
MUL			8.8		11.1
Average	3.7	3.8			
Maximum	5.3	5.4			
DIV	6.8	6.9	8.8		11.9
Unsuccessful	1.5	1.6	1.6		
Interrupt	1.8	2.2	1.6	3.0	5.2
Latency					12
With multiply-divide	9	9	10.6		
Without multiply-divide	5	5	4.6	6	
Data Channel					
Input	2.3	2.3	2.0	1.2	3.5
Output	2.8	2.8	2.0	1.2/1.8	4.4
Increment	2.8	2.8	2.2	1.8/2.4	4.4
Add	2.8	2.8			5.3
Latency			3.6		12
With multiply-divide	9	9			
Without multiply-divide	5	5		6	
High speed channel					
Input	.8	.8	.8		
Output	.8/1.0	.8/1.0	.8/1.0		
Increment	1.0/1.2	1.0/1.2	1.0/1.2		
Add	1.0/1.2	1.0/1.2			
Latency					
With IO	4.5	4.5	3.6		
Without IO	2.5	2.5	2.0		

‡ Add .3 if arithmetic or logical instruction is skipped, otherwise add .6.

APPENDIX E

IN-OUT CODES

The table on the next two pages lists the in-out devices, their octal codes, mnemonics and DGC option numbers. 8000 series options are the Supernova only, 8100 for the Nova 1200, 8200 for the Nova 800, and 4000 series options are for all machines or the Nova only. Codes 40 and above are used in pairs (40-41, 42-43, . . .) for receiver-transmitter sets in the high speed communications controller. The table beginning on page E4 lists the complete teletype code. The lower case character set (codes 140–176) is not available on the Model 33 or 35, but giving one of these codes causes the teletypewriter to print the corresponding upper case character. Other differences between the 33-35 and the 37 are mentioned in the table. The definitions of the control codes are those given by ASCII. Most control codes, however, have no effect on the computer teletypewriter, and the definitions bear no necessary relation to the use of the codes in conjunction with the software.

IN-OUT DEVICES

Octal Code	Mnemonic	Priority Mask Bit	Device	Option Number
01	MDV		Multiply-divide	A
02	MAP0		Memory allocation and protection	8008
03	MAP1			
04	MAP2			
05				
06	MCAT	12	Multiprocessor adapter transmitter	4038
07	MCAR	12	Multiprocessor adapter receiver	
10	TTI	14	Teletype input	4010
11	TTO	15	Teletype output	
12	PTR	11	Paper tape reader	4011
13	PTP	13	Paper tape punch	4012
14	RTC	13	Real time clock	4008
15	PLT	12	Incremental plotter	4017
16	CDR	10	Card reader	4016
17	LPT	12	Line printer	4018
20	DSK	9	Disk	4019
21	ADCV	8	A/D converter	4032 4033
22	MTA	10	Industry compatible magnetic tape	4033
23	DACV	–	D/A converter	4037
24	DCM	0	Data communications multiplexer	4026
25			Other multiplexers and/or control signal options	
26				
27				
30				
31*	IBM1	13	IBM 360 interface	4025
32	IBM2			
33				
34				
35				
36				
37				
40		8	Receiver	4015
41		8	Transmitter	
42				
43				
44				
45				
46				
47				
50			Second teletype input	4010
51			Second teletype output	
52			Second paper tape reader	4011

Octal Code	Mnemonic	Priority Mask Bit	Device	Option Number
53			Second paper tape punch	4012
54				
55				
56				
57				
60			Second disk	4019
61				
62			Second magnetic tape	4030
63				
64				
65				
66				
67				
70				
71* } 72			Second IBM 360 interface	4025
73				
74				
75				
76				
77	CPU		Central processor	B
			Power monitor and autorestart	C

*Code returned by INTA

A Supernova, 8007; Nova 1200, 8107; Nova 800, 8207; Nova, 4031

B Supernova, 8001; Nova 1200, 8101; Nova 800, 8201; Nova, 4001

C Supernova, 8006; Nova 1200, 8106; Nova 800, 8206; Nova, 4006

TELETYPE CODE

Even Parity Bit	7-Bit Octal Code	Character	Remarks
0	000	NUL	Null, tape feed. Repeats on Model 37. Control shift P on Model 33 and 35.
1	001	SOH	Start of heading; also SOM, start of message. Control A.
1	002	STX	Start of text; also EOA, end of address. Control B.
0	003	ETX	End of text; also EOM, end of message. Control C.
1	004	EOT	End of transmission (END); shuts off TWX machines. Control D.
0	005	ENQ	Enquiry (ENQRY); also WRU, "Who are you?" Triggers identification ("Here is. . .") at remote station if so equipped. Control E.
0	006	ACK	Acknowledge; also RU, "Are you. . .?" Control F.
1	007	BEL	Rings the bell. Control G.
1	010	BS	Backspace; also FEO, format effector. Backspaces some machines. Repeats on Model 37. Control H on Model 33 and 35.
0	011	HT	Horizontal tab. Control I on Model 33 and 35.
0	012	LF	Line feed or line space (NEW LINE); advances paper to next line. Repeats on Model 37. Duplicated by control J on Model 33 and 35.
1	013	VT	Vertical tab (VTAB). Control K on Model 33 and 35.
0	014	FF	Form feed to top of next page (PAGE). Control L.
1	015	CR	Carriage return to beginning of line. Control M on Model 33 and 35.
1	016	SO	Shift out; changes ribbon color to red. Control N.
0	017	SI	Shift in; changes ribbon color to black. Control O.
1	020	DLE	Data link escape. Control P (DC0).
0	021	DC1	Device control 1, turns transmitter (reader) on. Control Q (X ON).
0	022	DC2	Device control 2, turns punch or auxiliary on. Control R (TAPE, AUX ON).
1	023	DC3	Device control 3, turns transmitter (reader) off. Control S (X OFF).
0	024	DC4	Device control 4, turns punch or auxiliary off. Control T (AUX OFF).
1	025	NAK	Negative acknowledge; also ERR, error. Control U.
1	026	SYN	Synchronous idle (SYNC). Control V.
0	027	ETB	End of transmission block; also LEM, logical end of medium. Control W.
0	030	CAN	Cancel (CANCL). Control X.
1	031	EM	End of medium. Control Y.
1	032	SUB	Substitute. Control Z.
0	033	ESC	Escape, prefix. This code is also generated by control shift K on Model 33 and 35.
1	034	FS	File separator, Control shift L on Model 33 and 35.
0	035	GS	Group separator. Control shift M on Model 33 and 35.
0	036	RS	Record separator. Control shift N on Model 33 and 35.
1	037	US	Unit separator. Control shift O on Model 33 and 35.
1	040	SP	Space.
0	041	!	
0	042	"	

Even Parity Bit	7-Bit Octal Code	Character	Remarks
1	043	#	
0	044	$	
1	045	%	
1	046	&	
0	047	'	Accent acute or apostrophe.
0	050	(	
1	051	)	
1	052	*	Repeats on Model 37.
0	053	+	
1	054	,	
0	055	–	Repeats on Model 37.
0	056	.	Repeats on Model 37.
1	057	/	
0	060	Ø	
1	061	1	
1	062	2	
0	063	3	
1	064	4	
0	065	5	
0	066	6	
1	067	7	
1	070	8	
0	071	9	
0	072	:	
1	073	;	
0	074	<	
1	075	=	Repeats on Model 37.
1	076	>	
0	077	?	
1	100	@	
0	101	A	
0	102	B	
1	103	C	
0	104	D	
1	105	E	
1	106	F	
0	107	G	
0	110	H	
1	111	I	
1	112	J	
0	113	K	
1	114	L	
0	115	M	

Even Parity Bit	7-Bit Octal Code	Character	Remarks
0	116	N	
1	117	O	
0	120	P	
1	121	Q	
1	122	R	
0	123	S	
1	124	T	
0	125	U	
0	126	V	
1	127	W	
1	130	X	Repeats on Model 37.
0	131	Y	
0	132	Z	
1	133	[	Shift K on Model 33 and 35.
0	134	\	Shift L on Model 33 and 35.
1	135	]	Shift M on Model 33 and 35.
1	136	↑	
0	137	←	Repeats on Model 37.
0	140	`	Accent grave.
1	141	a	
1	142	b	
0	143	c	
1	144	d	
0	145	e	
0	146	f	
1	147	g	
1	150	h	
0	151	i	
0	152	j	
1	153	k	
0	154	l	
1	155	m	
1	156	n	
0	157	o	
1	160	p	
0	161	q	
0	162	r	
1	163	s	
0	164	t	
1	165	u	
1	166	v	
0	167	w	
0	170	x	Repeats on Model 37.

Even Parity Bit	7-Bit Octal Code	Character	Remarks
1	171	y	
1	172	z	
0	173	{	
1	174	|	
0	175	}	On early versions of the Model 33 and 35, either of these codes may be generated by either the ALT MODE or ESC key.
0	176	~	
1	177	DEL	Delete, rub out. Repeats on Model 37.

Keys That Generate No Codes

REPT	Model 33 and 35 only: causes any other key that is struck to repeat continuously until REPT is released.
PAPER ADVANCE	Model 37 local line feed.
LOCAL RETURN	Model 37 local carriage return.
LOC LF	Model 33 and 35 local line feed.
LOC CR	Model 33 and 35 local carriage return.
INTERRUPT, BREAK	Opens the line (machine sends a continuous string of null characters).
PROCEED, BRK RLS	Break release (not applicable).
HERE IS	Transmits predetermined 20-character message.

APPENDIX F

ASSEMBLY ERROR FLAGS

The listing output of assembly will contain alphabetic flags for errors occurring in assembly. The error flag appears on the lefthand side of the listing, beside the line containing the error. A given line may generate two or more error codes; for example, an undefined symbol appearing in an equivalence statement produces the flags U (undefined symbol) and E (equals error).

Flag	Error Example and Type of Error
A	STA 1,G (assume the STA instruction is in .ZREL coding and G is a symbol defined in .NREL coding.)
	A flags an addressing error indicating the address is out of the range for the type of assembly code. The error occurs on MRIs when the address is not within the range determined by the value of the index and/or is not within the current mode (.ZREL or .NREL) in relocatable assembly.
B	LA$L: LDA 1,23
	B flags a bad character; in the example, the $ in the label causes the error.
C	A+2:
	C flags a colon error. In the example, no expression is permitted to precede a colon.
D	.RDX 12
	D flags a radix error. In the example, radix 12 is not permitted.
E	REG= 3+B
	E flags an equals error. In the example, assume that B is undefined.
F	ADD 2
	F flags a format error. In the example, the ADD instruction requires at least two operands.
G	.EXTN S5
	G flags a symbol declaration error. In the example, the error would occur if S5 is not defined as .ENT in some extended assembly program communicating with the program containing the .EXTN.

Flag	Error Example and Type of Error
I	This error occurs on a parity check of input of the source program. The assembler flags the line and substitutes a back slash for the character in error.
K	This error occurs on .IFE and .IFN pseudo-ops. It indicates either that the expression in the pseudo-op cannot be evaluated during pass 1 of assembly or that a second .IFE or .IFN has been encountered before an .ENDC pseudo-op. This is taken as an illegal attempt to nest conditional assemblies.
L	.LOC -1 (bit Ø is set.) L flags a location error and can occur on .LOC and .BLK pseudo-ops, for example, an attempt to set the location counter back or set it to an address outside the range of memory.
M	A: 3 A: 4 M flags a multiply-defined symbol.
N	C77: 7A N flags a number error. In the example, 7A is not a number, since no letters are permitted.
O	LDA 4,.LOC O flags a field overflow -- an accumulator greater than 3, as in the example, a skip field greater than 7; etc.
P	P flags a phase error in which the value of a symbol on pass 2 of assembly differs from its value on pass 1.
Q	.+.END Q flags a questionable line of any type not specifically defined for another error code.
R	J: C+F (assume C is page zero and F normal relocatable.) R flags expression errors occurring in relocatable assembly when the expression cannot be evaluated to be absolute, relocatable, or byte pointer type relocatable; or when page zero and normal relocatable symbols are mixed in an expression and

neither the page zero nor normal relocatable symbols are canceled out when the expression is evaluated.

S	S flags the overflow of the symbol table, occurring when memory capacity for the particular machine has been reached.

T	14+.XPNG

T flags an error occurring in a symbol table pseudo-op. In the example, no expression is permitted to precede the pseudo-op .XPNG.

U	LDA 2,B	(assume B is undefined.)

U flags an undefined symbol. Symbols that will be flagged undefined are:
  A symbol whose value is not known on pass 2.
  A symbol in an expression that must be evaluated in pass 1.

X	LET: "C3
3+.TXT

X flags a text error. In the first example, only a single character may follow "; in the second example, the .TXT pseudo-op cannot be preceded by an expression.

Z	LDA 3,DISP+6	(assume DISP appears in a .EXTD in the program.)

Z flags certain illegal symbols appearing in expressions in relocatable assembly code: externals, op-codes, double precision numbers, and floating-point numbers.

!
    in debugging command 7-2
    (OR) 3-2

" special character 3-2, 3-3

special character 3-2 to 3-4

$
    in editing command 4-3
    in debugging command 7-2

& (AND) 3-2

.
    editing command 4-3
    special character 3-2, 3-3
    in label 3-1

:
    after label 3-1
    editing command 4-3

=
    Debug II command 7-5
    editing command 4-3

@ special character 3-2, 3-3

↑ in debugging command 7-2

A
    editing command 4-1
    error flag F-1
    Debug II command 7-1

ABSOLUTE addressing 9-1

ACCUMULATOR
    examine/modify under Debug 7-1
    general 1-1
    in ALI instructions 2-6
    in I/O instructions 2-11

ACCUMULATOR (con'd)
    in MRI instruction 2-1
    switches 5-2

ACD (destination AC) 2-7

ACS (source AC) 2-7

A/D converter (ADC) E-2

ADC 2-8, D-5

ADD 2-8, D-5

ADD COMPLEMENT instructions (ADC_) 2-8, D-5

ADDITION instruction (ADD_) 2-8, D-5, D-6

ADDRESS
    examining an _ 7-2
    indicators 5-1
    modifying an _ 7-2

ADDRESSING
    absolute 9-1
    device register 2-2
    direct 2-2
    error F-1
    indirect 2-2
    mode field of MRI 2-1
    normal relocatable 9-1
    relative 2-2
    use of @ sign 2-3
    zero relocatable 9-1

ALC (see arithmetic and logical instructions)

ALIGNMENT, bit 9-9

AND 2-8, D-6

ANDING instruction (AND_) 2-8, D-6

ARITHMETIC AND LOGICAL instructions
definition 1-2
discussion 2-6 to 2-11
functional portion 2-7
functions of 2-8
processing 2-7

ARITHMETIC EXPRESSION
calculations under Debug II 7-5
permitted by assembler 3-2

ASCII (teletype) codes E-4

ASSEMBLER
Basic Chapter 3, B-11
binary output devices 3-12
extensions to relocatable 9-8
input devices 3-11
listing devices 3-11
operating mode 3-12
operating procedures 3-11
Relocatable Chapter 9, B-1

ASSEMBLY, conditional 9-10

AT SIGN (@) 2-3, 3-2, 3-3

AUTO-DECREMENTING locations 2-6

AUTO-INCREMENTING locations 2-6

AUTO-RESTART 8-5, E-3

B
Debug II command 7-3
editing command 4-2
error flag F-1

BIN: 3-12

BINARY
-decimal conversion routine A-6
-octal conversion routine A-5

BIT BOUNDARY alignment 9-9

.BLK
error F-2
pseudo-op 3-5, C-1, 9-1 ff

BREAKPOINT 7-1, 7-3

BROADCASTING DS code 8-3, 8-4

BUFFER
I/O 2-11
editing 4-1

BUSY
flip flop 2-12 to 2-14, 8-1

BYTE
manipulation 10-1
swapping 2-10

C
carry bit 2-9, 3-10
control function 2-12, 3-10, 8-3
Debug II command 7-2
editing command 4-2
error flag F-1

CARD READER (CDR) E-2

CARRY
base value 2-8, 2-9
field 2-9
general 1-1
value on overflow 2-8

CARRIAGE RETURN (CR)
end of line 3-2
in character string 3-5
use in debugging command 7-2

CENTRAL PROCESSOR (CPU) E-3

CHANNEL START switch 5-3

CHARACTER
error in F-1
pointer 4-1
string 3-6
special 3-2
. 3-2, 3-3
" 3-2, 3-3
@ 3-2, 3-3
# 3-2, 3-3, 3-4
teletype input routine A-2
teletype output routine A-3

COLON (:)
error F-1
usage 3-1

COM 2-8, D-6

COMMA separator 3-1

COMMAND field of MRI 2-1

COMMENT 3-1, 3-2

COMPLEMENT instructions (COM_) D-6, D-7, 2-8

CONDITIONAL assembly
error F-2
pseudo-ops 9-10

CONSOLE
indicators Cahpter 5
Supernova 5-4
switches Chapter 5

CONTINUE switch
after .EOT 3-6
on console 5-2

CONTROL function 2-11, 2-12

CONVERSION
octal-binary routine A-4
binary-octal routine A-5
binary-decimal routine A-6

CORE storage
amount 1-1
modification 2-4

CP (character pointer) 4-1

CPU
code E-3
instructions Chapter 2

D
displacement field 2-1
error code F-1
editing command 4-2
Debug II command 7-4

D/A converter (DACV) E-2

.DALC 3-9, C-1

DATA
indicators 5-1
I/O transfer 2-11, 8-5
manual entry of 1-1
moving instructions 2-3
transfer lines 8-3, 8-5

DATA COMMUNICATIONS multiplexer (DCM) E-2

DATA IN instructions (DI_) D-7, 2-14

DATA OUT instructions (DO_) D-7, 2-13

DCH
indicator 5-1
interrupt 8-5

DEBUGGING
Debug I 7-1, B-1
Debug II Chapter 7, B-1
Debug III 7-1, B-2, 9-11

DECREMENT and skip if zero instruction (DSZ) 2-4, D-7

DEFER indicator 5-1

DEPOSIT switch 5-2

DEPOSIT NEXT switch 5-2

DEVICE
- code field 2-11
- code 77 2-14
- octal codes for Appendix E
- selection lines 8-3
- service code (DS) 8-3, 8-4

.DIAC 3-10, C-1

DIAGNOSTICS Appendix F

.DIO 3-10, C-1

.DIOA 3-10, C-1

DISK (DSK) E-2

DISPLACEMENT external symbol 9-4

DISPLACEMENT field of MRI 2-1

DIVIDE instruction (DIV) D-7

.DMR 3-8, C-1

.DMRA 3-9, C-1

DONE
- flip flop 2-12 to 2-14, 8-1, 8-2, 8-3

DOUBLE PRECISION numbers 9-8

DSZ 2-4, D-7

DUMP routine A-7

.DUSR 3-7, C-1

E
- Debug II command 7-4
- effective address 2-1
- error code F-1

EDITOR Chapter 4, B-2

EFFECTIVE ADDRESS
- definition 2-1
- range 2-2

.END 3-7, C-1

.ENDC 9-10, C-1

.ENT 9-4, C-1

ENTRY POINT of subprogram 9-4

.EOT 3-7, C-1

EQUIVALENCE statement
- example 3-4
- error in F-1

ERROR FLAGS Appendix F

EXAMINE switch 5-2

EXAMINE NEXT switch 5-2

EXECUTE indicator 5-1

EXECUTION TIMING D-12

EXPRESSION
- arithmetic 3-2
- error in F-2

.EXTD 9-4, C-2

.EXTN 9-4, C-2

F
- Debug II command 7-4
- editing command 4-2
- error flag F-1

FETCH indicator 5-1

FIELD
- ALI 2-7, 2-8
- assembler 3-1
- I/O 2-11
- MRI 2-1

FLIP FLOP
- Busy 2-12, 8-1
- Done 2-12, 8-1, 8-2
- Interrupt Disable 8-1, 8-2
- Interrupt Request 8-1

FLOATING POINT interpreter 9-8, B-2

FORM FEED (FF) 3-2

FORMAT
- assembler program 3-2
- error F-1

FUNCTION GENERATOR 2-7

FUNCTION OF ALI 2-7 to 2-11

G error flag F-1

GLOBAL symbol 9-4

HALT 2-14, D-8

HARDWARE INTERRUPT
- locations used for 2-6
- handling of Chapter 8

I
- addressing mode bit 2-2
- editing command 4-2
- error flag F-2

IBM 360 interface E-2

.IFE 9-10, C-2

.IFN 9-10, C-2

IN: 3-11

INC 2-8, D-8

INCREMENT and skip if zero instruction (ISZ) 2-4, D-8

INCREMENT instructions (INC_) 2-8, D-8

INCREMENTAL PLOTTER (PLT) E-2

INDEX field of MRI 2-1

INDICATOR 1-1, 5-1

INPUT
- instructions 2-11
- parity error F-2
- to assembler 3-11

INITIALIZATION routine A-17

INPUT/OUTPUT
- instruction
  - definition 1-2
  - discussion 2-11 to 2-14
- device codes Appendix E

INST. STEP switch 5-2

INSTRUCTION
- execution time D-12
- indicators 5-1
- mnemonic derivations D-1
- set, octal listing D- 2
- set, alphabetical listing D-5

INTA instruction 8-3, D-8

INTDS 8-1, 8-2, D-8

INTEN 8-1, 8-2, D-8

INTERRUPT
- acknowledge instruction (INTA) 8-3, D-8
- disable
  - flip flop 8-1, 8-2
  - instruction (INTDS) 8-1, 8-2, D-8
- enable instruction (INTEN) 8-1, 8-2, D-8
- enabling 8-1, 8-2
- levels of 8-2
- of program by device Chapter 8
- request flip flop 8-1
- routine to ignore A-16
- servicing Chapter 8

INTP (Interrupt priority line) 8-3

INTR (Interrupt Request Signal) 8-1, 8-3

I/O bus 8-3

ION indicator 5-1, 8-1, 8-2, 8-3

IORST (I/O Reset) instruction 8-4, D-8

ISZ 2-4, D-8

J editing command 4-2

JMP 2-4, D-8

JSR 2-4, D-8

JUMP instruction (JMP) 2-4, D-8

JUMP subroutine instruction (JSR) 2-4, D-8

K
- editing command 4-2
- error flag F-2

L
- editing command 4-2
- error flag F-2, 9-3
- register in debugging 7-4
- shift bit 2-10, 3-10

LABEL 3-1

LDA 2-3

LINE
- editing a 4-1
- error in F-2
- of listing page 3-2

LINE FEED (LF)
- ignored as symbol 3-11
- use in debugging command 7-2

LINE PRINTER (LPT) E-2

LIST: 3-11

LISTING
- assembler 3-2, 3-11
- under debugger 7-3

LOAD ACCUMULATOR instruction (LDA) 2-3, D-8

LOAD/NOLOAD field 2-10, 2-11

LOADING
- Binary Loader 6-2, B-3
- Bootstrap Loader 6-1, B-3
- Nova 800/1200 Selfload Bootstrap 6-2, B-3
- Relocatable Loader 9-2, B-3
- Supernova Selfload Bootstrap 6-2, B-3

.LOC
- error F-2
- pseudo-op 3-4, C-2, 9-1 ff

LOCAL symbol 9-11, 3-12

LOCK switch 5-1

M
- editing command 4-2
- error flag F-2
- mask register 7-2

MAGNETIC TAPE (MTA) E-2

MASK OUT instruction (MSKO) 8-2, D-9

MEMORY (see also core storage)
modification instructions 2-4
search in debugging 7-2

MEMORY ALLOCATION and protection (MAP_) E-2

MEMORY REFERENCE instructions
definition 1-1
discussion 2-1 to 2-6

MEMORY STEP switch 5-2

MODE: 3-12

MODE OF ASSEMBLY 3-12

MODIFY MEMORY
instructions 2-4
under debugger 7-2

MOV 2-8, D-8

MOVE instructions (MOV_) 2-8, D-8, D-9

MOVE DATA instructions 2-3

MRI (See memory reference instructions)

MSKO instruction 8-2, D-9)

MULTIPLE DEFINITION error F-2

MULTIPLEXERS E-2

MULTIPROCESSOR adapter (MCA) E-2

MULTIPLY DIVIDE option (MDV) E-2

MULTIPLY instruction (MUL) D-9

N error flag F-2

NEG 2-8, D-9

NEGATE instructions (NEG_) 2-8, D-9

NO I/O TRANSFER instruction (NIO_) D-9, D-10, 2-13, 2-14

NORMAL EXTERNAL symbol 9-4

NORMAL RELOCATABLE code 9-1

.NREL 9-1 ff, C-3

NULL
ignored as symbol 3-11
in character string 3-5

NUMBER error F-2

O
ALI carry bit 2-9, 3-10
error flag F-2

OBJECT TAPE 3-12

OCTAL BINARY conversion routine A-4

OFF switch 5-1

ON switch 5-1

OPCODE 3-1

OPERAND 3-1

OUTPUT
binary from assembly 3-12
instructions 2-11
listing 3-2, 3-11

OVERFLOW
carry value on 2-8
conditions causing 2-9
error F-2
I/O bus line 8-6
of symbol table F-3

OVERLAP indicator 5-1

P
control function 2-12, 3-10, 8-3
Debug II command 7-4
editing command 4-1
error flag F-2

PACKING character string 3-6

PAGE
editing 4-1
listing 3-2

PAPER TAPE punch (PTP) E-2

PAPER TAPE reader (PTR) E-2

PARITY
character string 3-6
requesting checking 3-11

PERIOD (.)
in label 3-1
special character 3-2, 3-3

PHASE ERROR F-2

PI indicator 5-1

PLOTTER (PLT) E-2

POLLING DS requests 8-3

POUND SIGN (#) 3-2 to 3-4

POWER MONITOR and auto-restart 8-5, E-3

POWER switch 5-1

PRE-DEFINED MEMORY cells 2-6

PRIORITY
device 8-1
determining 8-3

PROGRAM COUNTER 1-1

PROGRAM
examples Appendix A
sequence
changing the 2-4
normal 1-1
tips Chapter 11

PROGRAM LOAD switch 5-2

PROTECT indicator 5-2

PSEUDO-OP 3-4, Appendix C

PW editing command 4-2

Q error flag F-2

QUOTATION MARK (") 3-2, 3-3

R
Debug II command 7-4
editing command 4-2
error flag F-2
shift bit 2-10, 3-10

RADIX
changes 3-5
error F-1

.RDX 3-5, C-2, 9-8

READS 2-14, D-10

REAL TIME clock (RTC) E-2

RELOCATABLE
    coding   Chapter 9
    symbol error   F-3

RELOCATABLE LOADER   9-2

RELOCATION PSEUDO-OP   9-1 ff

RESET switch   5-2

RUBOUT
    in character string   3-5
    ignored as symbol   3-11

RUN indicator   5-1

S
    control function   2-12, 3-10, 8-3
    editing command
    error flag   F-3
    shift bit   2-10, 3-10

SAMPLE PROGRAMS   Appendix A

SBN   2-10

SEMICOLON (;)   3-2

SEZ   2-10

SHIFT
    byte -   10-1
    shifter   2-7
    field   2-10

SIGN   2-11

SKIP
    field   2-10
    sensor   2-7
    list of mnemonics   D-1
    instruction (SKP_)   2-13, D-10

SKP
    field   2-10
    instructions   2-13, D-10

SKPBN   2-13, D-10

SKPBZ   2-13, D-10

SKPDN   2-13, D-10

SKPDZ   2-13, D-10

SNC   2-10

SNR   2-10

SPACE separator   3-1

SPECIAL characters   3-2 to 3-4

STA   2-3, D-10

START switch   5-2

STOP switch   5-2

STORE ACCUMULATOR instruction (STA)   2-3, D-10

SUB   2-8, D-10

SUBTRACT instructions (SUB_)   2-8, D-10

SWITCHES, console   Chapter 5

SYMBOL
    declaration error   F-1
    error   F-3
    external displacement   9-4
    global   9-4
    local   3-12, 9-11
    multiple definition error   F-2
    normal displacement   9-4
    relocatable assembly - error   F-3

SYMBOL TABLE
    overflow error   F-3
    pseudo-op   3-7 ff
    pseudo-op error   F-3

SYMBOLIC DEBUGGER 9-11, 7-1, B-1

SZC 2-10

SZR 2-10

T
- editing command 4-1
- error flag F-3

TAB
- separator 3-1
- tabulation 3-2

TELETYPE
- character 10-1
- character codes E-5
- character output routine A-3
- character reading routine A-2
- device codes for E-2
- models E-1

TERMINATE source program 3-6

TEXT
- Editor Chapter 4, B-2
- error in F-3
- pseudo-ops 3-6

TIMING of execution D-12

TIPS, programming Chapter 11

.TITL 9-4, C-2

TRANSFER
- field 2-11, 2-12
- special code 7 2-13

TTI E-2

TTO E-2

.TXT 3-6, C-2

.TXTE 3-6, C-2

.TXTF 3-6, C-2

.TXTM 3-6, C-2

.TXTO 3-6, C-2

U error flag F-3

W(word) register 7-2

WORD, Nova 1-1

X
- editing command 4-2
- error flag F-3
- index 2-1, 2-2

XD editing command 4-2

XM editing command 4-2

.XPNG 3-11, C-2

Y editing command 4-1

Z
- ALI carry bit 2-9, 3-10
- editing command 4-2
- error flag F-3

ZERO relocatable code 9-1

.ZREL 9-1 ff, C-2

cut along dotted line

DATA GENERAL CORPORATION
PROGRAMMING DOCUMENTATION
REMARKS FORM

DOCUMENT TITLE ______________________________

DOCUMENT NUMBER (lower righthand corner of title page) ______________

Specific Comments. List specific comments. Reference page numbers when applicable. Label each comment as an addition, deletion, change or error if applicable.

General Comments and Suggestions for Improvement of the Publication.

FROM: Name: ______________________ Date: __________
Title: ______________________
Company: ______________________
Address: ______________________
____

FOLD DOWN FIRST FOLD DOWN

FIRST
CLASS
PERMIT
No. 26
Southboro
Mass. 01772

BUSINESS REPLY MAIL

No Postage Necessary If Mailed In The United States

Postage will be paid by:

Data General Corporation

Southboro, Massachusetts 01772

ATTENTION: Programming Documentation

FOLD UP SECOND FOLD UP

STAPLE

DATA GENERAL
CORPORATION
Southboro,
Massachusetts 01772
(617) 485-9100