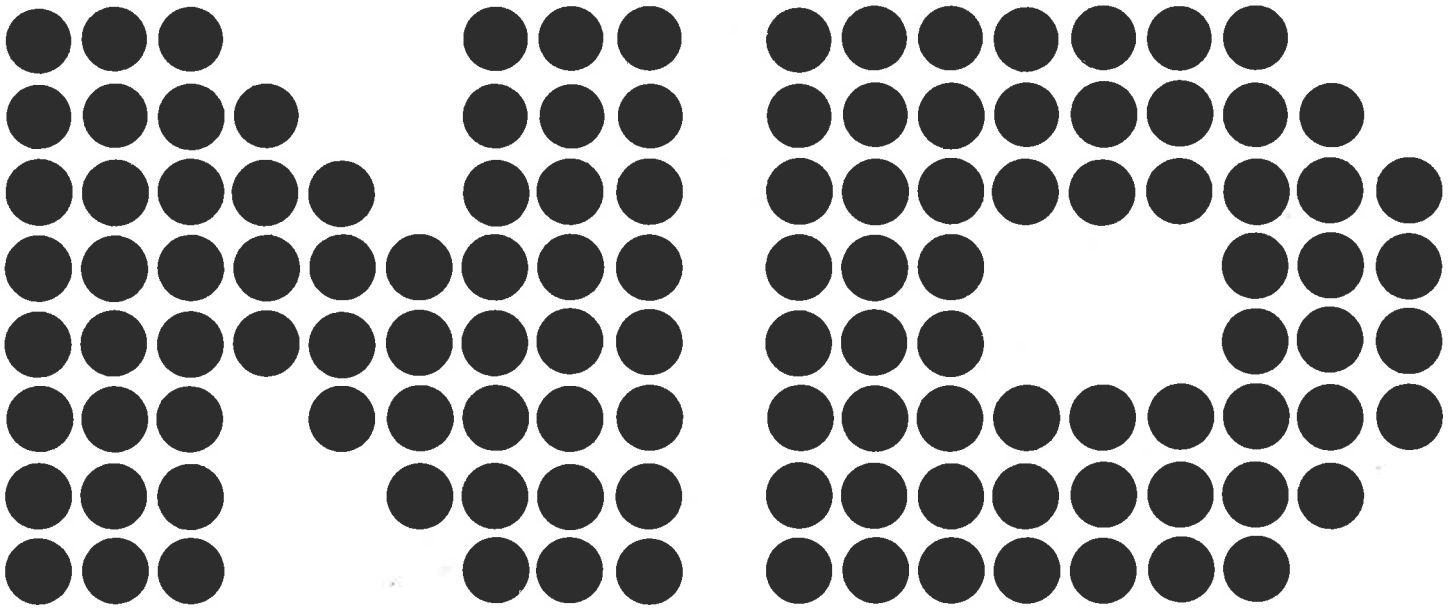
MAC

nteractive Assembly and Debugging System
User's Guide

NORSK DATA A.S



MAC

Interactive Assembly and Debugging System
User's Guide

REVISION RECORD

Revision	Notes
03/78	NEW ISSUE
	Replaces previous issue numbered ND-60.001.01.

MAC — Interactive Assembly and Debugging **System, User's Guide**
ND-60.096.01

NORSK DATA A.S.
Lørenveien 57, Postboks 163 Økern, Oslo 5, Norway

TABLE OF CONTENTS
+ + +

1 INTRODUCTION

1.1 *PHILOSOPHY OF MAC*

MAC has the capability to accept code in the MAC language, the symbolic assembly language for the NORD-10 computer, and to assemble the MAC code into binary machine code as is necessary for program execution. MAC also has the capability to examine and change a program, once the program has been assembled and loaded, and to perform many other functions normally more closely associated with an interactive debugging system.

Furthermore, MAC has a simple program editing capability. MAC was designed so it can always be memory resident with the capability to assemble programs directly into memory. Consequently, MAC can be loaded and then continually used for all phases of the program construction process. Thus, the MAC Interactive Assembly and Debugging System has proved to be very powerful and convenient.

The concept of MAC is not much like that of traditional assemblers. It is a survivor in the history of Norsk Data and remains active after alot of adjustments and improvements. One important thing to be emphasized in the introduction of this manual; the concept of MAC allows the user to be extremely free in his composition of assembly source programs as well as in the interaction while debugging programs. We at Norsk Data consider this as an advantage, even if the lack of a strict syntax may lead to mistakes and problems for the inexperienced user.

The necessity of having a machine-oriented language will always be present irrespective of the EDP application. However, program systems are becoming more and more complex, thus being difficult to develop and maintain. In general, but also at Norsk Data, this has enforced so-called machine-oriented and problem-oriented (high level) languages which produce optimal machine code. Such languages are often implemented as "pre-processors" to the actual assembler languages because including of assembly code sequences is very easy.

The machine-oriented language, NORD Programming Language (NORD PL or NPL) produces an object output which is MAC assembler source code. Hence, all the debug facilities of MAC are immediately available, including symbolic references to lables and variables. The binary object code produced by all NORD language processors is called Binary Relocatable Format (BRF). This standard binary format may, of course, also be produced by MAC.

The intention is obvious; an uncomplicated way to mix languages of any source.

The MAC system has developed into some different versions which have the basic MAC principles in common. To avoid confusion we shall give a short description in the following lines:

MAC

Standard assembler, SINTRAN III subsystems or stand-alone.

MACF (MAC-File)

SINTRAN III subsystem. Absolute assembly goes to a memory image-file. Special features are documented in Appendix C of this manual.

MACD or DMAC (MAC-Debugger)

Debugging system under SINTRAN III. Special features are documented in the manual "SINTRAN III User's Guide".

MACM (MAC-Mass Storage)

Stand-alone system. Absolute assembly goes directly to different mass storage devices. Mainly used to generate and start SINTRAN III. Special features are documented in the manual "MACM - MAC Mass Storage Assembler". A short description is given in Appendix C of this manual.

1.2 *THE STRUCTURE OF THIS USER'S GUIDE*

This User's Guide is intended to serve as an introduction to the philosophy, use, and functioning of the MAC Interactive Assembly and Debugging System for the NORD-10 computer. It is also intended to serve as a comprehensive operation, maintenance, and reference manual. The guide does not document the internals of the MAC system.

It is assumed that readers of this "User's Guide" have a rudimentary understanding of assembly language programming and of word-oriented, single-address computers. A specific example in the latter area is presented in another volume, "The NORD-10 Reference Manual".

This manual is divded into six chapters, with this introduction as Chapter 1. In Chapter 2, the hardware environment in which MAC resides and for which MAC assembles code is discussed. This chapter includes a thorough discussion of the NORD-10's addressing structure and instruction repertoire. In Chapter 3, the basic MAC system is described, first in a simplified, intuitive fashion, and then rigorously. Chapter 4 describes the various options that can be added to MAC. Chapter 5 leads the reader through a set of examples which illustrate the steps of writing, assembling, and debugging a program using MAC. And finally, Chapter 6 gives an introduction to subroutines and transfer of parameters.

In addition to these main chapters, this guide has a number of appendixes which include a list of all MAC's built in symbols and a complete list of error messages. Finally, an index is provided. Throughout the text of the guide, appropriate cross-references are given.

Readers with only a rudimentary background in assembly language programming are advised to read carefully Chapter 2, Sections 3.1, 3.2.1, 3.2.2, and Chapter 5, and to then write and debug a small program before attempting to read the rest of this guide. Experienced assembly language programmers should be able to start coding after glancing over the Table of Contents and Sections 3.1, 3.2, and 4.2. The index will be of aid to all users.

Some documentation conventions and terms which may seem confusing are explained below:

- capital letters marked with a like A or Q indicate the respective key on the keyboard plus the CTRL key.
- Elements may be described in general terms by enclosing a self-explanatory text in .
- Terminal is any device having a two-way communication with the computer.
- The user types on the keyboard and MAC prints on the terminal.

1.3 *REVISIONS AND CORRECTIONS*

This guide represents the amalgamation of all preceding MAC related documents and supersedes them all. Future changes to the MAC system will be documented in a series of revisions to this user's guide which will be issued to coincide with the release of revisions to the MAC system. Also, revisions will be issued as necessary to correct errors in the guide or to improve the presentation.

The "loose-leaf" format of the guide will facilitate its frequent revision, since only pages which have been changed need be distrubuted. Frequent revision will insure users up-to-date information about the MAC system. The changed pages distributed as a revision will be dated on the revision page so the user can correctly up-date his own copy of the guide.

This is your guide. You know best what it should contain. Although we at Norsk Data have tried to anticipate your needs, great improvement in the guide's usefulness and clarity can be achieved if you send us your comments about it, corrections to it, and suggestions for its improvement will be very much appreciated. Use the pre-addressed "Comment and Evaluation Sheet" at the back of this manual.

2 THE HARDWARE ENVIRONMENT FOR MAC

The purpose of the MAC language, as with all assembly languages, is to symbolically represent words in a computer, in this case the binary words of the NORD-10 computer. Therefore, before the MAC language is described, it is appropriate to introduce the reader to the data and instruction formats of the NORD-10 computer. A more complete description of the NORD-10 computer may be found in the "NORD-10 Reference Manual".

The configuration for operation of MAC ranges from the most advanced SINTRAN III installation to the stand-alone NORD-10 with a console typewriter. Each addition of input/output devices will, of course, enhance MAC's performance.

2.1 *INSTRUCTION AND DATA FORMATS IN THE NORD-10*

The NORD-10 has a 16-bit word. The bits are conventionally numbered 0 to 15 with the most significant bit numbered 15 and the least significant numbered 0.

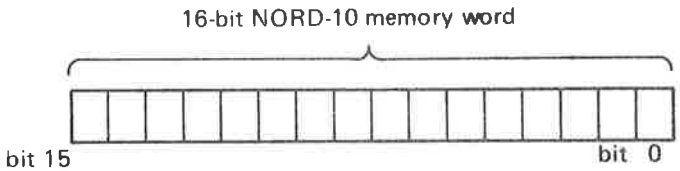


Figure 2.1: NORD-10 Bit Number Convention

The content (or value) of a memory word is conventionally represented by a 6-digit octal number. Thus, the content of a memory word with all 16 bits set to zero is represented as 000000, while the content of a memory word with all bits set to one is represented as 177777.

All instructions are contained in exactly one memory word, while data items contained in one, two or three consecutive memory words are recognized by the NORD-10 Central Processor Unit (CPU).

2.1.1 *Single Bit*

A single bit data word is typically used for a logical variable; the bit instructions are used for manipulation of single bit variables. The bit instructions specify operations on any bit in any of the general registers, as well as the accumulator indicator K.

2.1.2 *Half Word Data Items (Bytes)*

The internal representation of characters is ASCII which is an 8-bit code including a parity bit. In character strings such bytes are packed two by two in one 16-bit word. The NORD-10 provides instructions to handle bytes. The even byte address points to the left half of the word.

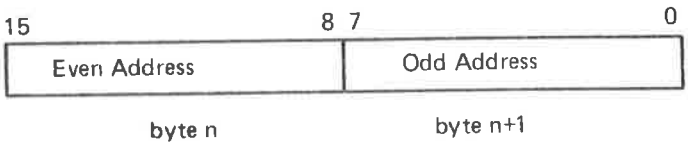


Figure 2.2: NORD-10 Byte Format

2.1.3 *Single Word Data Items*

Single word data items are considered by the NORD-10 to be either integer arithmetic or Boolean (logical) 16-bit numbers, and instructions exist to operate on both types of numbers. Negative arithmetic numbers are represented in two's complement notation, and if the arithmetic numbers are considered to be signed integers, the range of possible integer values is from −32768 to 32767. Alternatively, integer numbers are often considered to range from 0 to 65537 - except for overflow indications, two's complement arithmetic gives the correct answer whether numbers are considered to have a sign bit and 15 bits of magnitude or to have no sign and 16 bits of magnitude.

2.1.4 *Double Word Data Items*

The optional 32-bits floating point format is discussed in Appendix E.

The programmer may consider two-word data items to be of any type he desires, since the standard NORD-10 has no hardware instructions which operate on these data items other than load and store instructions. However, it is sometimes convenient to consider two-word data items to be 32-bit numbers, with the more significant 16 bits residing in one memory location and the less significant 16 bits residing in the next higher numbered memory location. If this 32-bit number is considered to be a double integer with negative values represented using two's complement notation, the possible range of integer values is −2 147 483 648 to 2 147 483 647.

A double word is always referred to by the address of its most significant part. Normally, a double word is transferred to the registers so that the most significant part is contained in the A register and the least significant in the D register.

2.1.5 *Triple Word Data Items*

The NORD-10 computer uses three-word data items to hold standard floating point numbers, also called real numbers.

The data format of floating point words is 32 bits mantissa magnitude, one bit for the sign of the number and 15 bits for a signed exponent.

The mantissa is always normalized, $0.5 \leq$ mantissa < 1; for all non-zero numbers bit 31 equals one. The exponent base is 2. The exponent is biased with 2^{14}, i.e. 40000_8 is added to the actual exponent, so that a standardized floating zero contains zero in all 48 bits.

In the computer memory one floating point data word occupies three 16 bit locations, which are addressed by the address of the exponent part.

n	exponent and sign
n + 1	most significant part of mantissa
n + 2	least significant part of mantissa

In CPU registers, bits 0-15 of the mantissa are in the D register, bits 16-31 in the A register, and bits 32-47, exponent and sign, in the T register. These three registers together are defined as the floating accumulator.

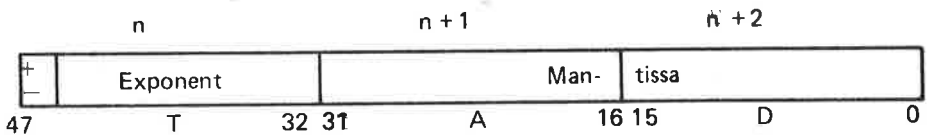


Figure 2.3: NORD-10 Standard Floating Point Format

The accuracy is 32 bits or approximately 9 decimal digits, any integer up to $2^{32}-1$ has an exact floating point representation. The range is:

$2^{-16384} * 0.5 \leq |x| < 2^{16383} * 1$ or $x = 0$

or

$10^{-4931} < |x| < 10^{4931}$

Examples (octal format):

	T	A	D
0:	0	0	0
+1:	040001	100000	0
−1:	140001	100000	0

Incidently, the instructions for loading and storing two and three word data items are often used as a quick method of simultaneoulsy loading or storing two or three single word data items.

2.1.6 *Instructions*

All instructions have an operation code in the most signifcant (left) five bits of the computer word, the bits set in the octal number 174000; therefore, there are 32 basic instructions in the NORD-10.

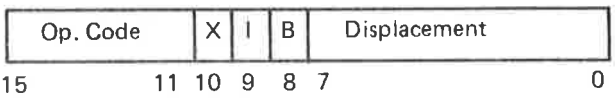


Figure 2.4: NORD-10 Instruction Word

The first 24 instructions are commonly called memory address instructions, and in these instructions, bits 0-7 contain a signed 7-bit number called the displacement. The displacement is always considered to be relative (−128 to 127) to the content of the X, B or P register. In 23 of the first 24 instructions the content of bits 10, 9 and 8 (represented by 3400_8) select one of 8 addressing modes. The exception is the conditional jump instruction (CJP) which uses these bits to specify one of 8 sub-instructions. Also, the displacement in the CJP instruction is always relative to the P register.

Of the remaining eight basic instructions, one is an input/output group and the other specifies what is often called operated groups (i.e., instructions which do not reference memory locations or do input/output, but do shifts, inter-register arithmetic, register testing, etc.) The sub-instructions in these groups are illustrated later.

2.2 *THE NORD-10 ADDRESSING STRUCTURE*

As stated in the previous section, a thorough understanding of the hardware environment for which MAC assembles code is essential for a thorough understanding of MAC. Therefore, in this section we try to give the user some understanding of the use of the NORD-10 addressing structure. Wise use of the addressing structure will usually result in significant efficiencies in a program. Advanced readers may want to skip this section after looking at the table in Appendix D which summarizes the addressing structure.

An important preliminary to this discussion of the addressing structure is an explanation of the concept of an effective address, a concept relevant only to memory address instructions. The effective address is the address of the memory location that is finally accessed after all address modifications have taken place in memory address instructions. Suppose, for example, that a particular instruction is executed which results in the content of the X register being added to the displacement and finally the content of memory location 14030 being loaded into the A register. The effective address of this particular execution of this particular instruction is 14030.

The NORD-10 has eight different addressing modes, any one of which may be selected using the X, I and B bits in all memory address instructions except the conditional jump instruction. Explanations of these eight modes follow.

2.2.1 *P Relative Addressing*

The first mode which we shall describe is called the *P relative* addressing mode and is specified by setting the X, I and B bits all to zero. In this mode the displacement bits (bits 0-7) specify a positive or negative 7-bit address relative to the current value of the instruction counter (P register).

Suppose memory location 403 contains the instruction 004002_8 which in this chapter we shall represent by STA 2 and this instruction is executed. (Note that this is not the way the instruction is written in the MAC language.) The X, I and B bits are all set to zero indicating P relative addressing, and a positive displacement of 2 is given; therefore, the contents of the A register will be stored in memory location 405. If instead location 403 contains the instruction JMP −2 and it is executed, the next instruction to be executed will be taken from location 401. While there is an obvious limitation to this mode of addressing (locations more than 128_8 words away from the instruction being executed cannot be accessed), this mode of addressing is still quite useful for doing local jumps and accessing nearby constants and variables.

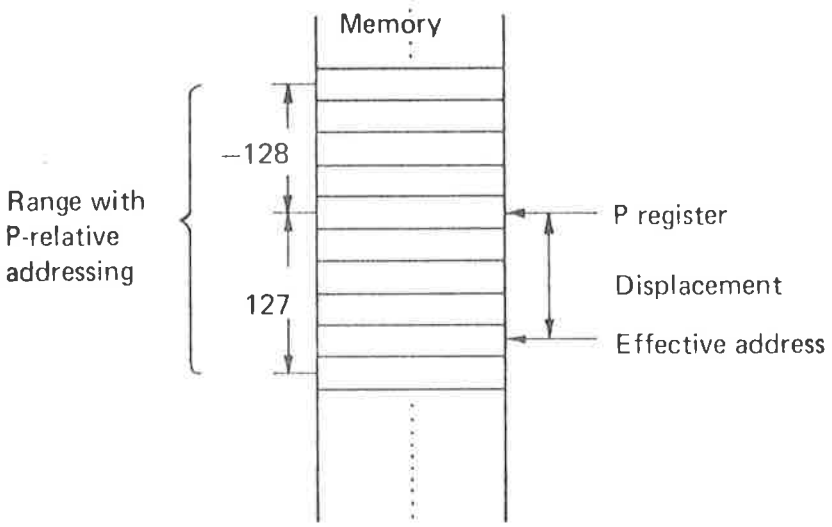


Figure 2.5: Schematic Illustration of P Relative Addressing

2.2.2 *Indirect P Relative Addressing*

Since one must be able to access memory locations more than 128_8 words away from the instruction being executed, the simplest method of doing this is to use the *indirect P relative* addressing mode, specified by setting the I bit to one and the X bit and B bit to zero in memory address instructions. In this mode an address relative to the program counter is computed, just as for P relative addressing, by adding the displacement to the value of the program counter; but then, rather than the addressed location actually being accessed, the contents of the addressed location are used as a 16-bit address of a memory location which is accessed instead. The following example will make this clearer.

Suppose location 405 contains the instruction LDA I 2 (045002_8) and this instruction is executed. Further, suppose memory location 407 contains the value 16003 and memory location 16003 contains the value 17. The net result of executing the instruction in location 405 is to load the value 17 in the A register. First the displacement, 2, of the LDA instruction is added to the value of the location counter, 405, giving the result of 407; then the contents of location 407, 16003, are used as an address and the contents of this address, 17, is finally loaded into the A register.

An analogy may also be helpful. Suppose your company's mail boy is told to pick up a letter from an office, the identity of which may be found on a piece of paper in the second office, beyond the office where the mail boy currently is. Substitute memory location for office in this analogy and you have indirect P relative addressing. This addressing mode obviously allows access of locations anywhere in a 64K (full 16-bit address) memory.

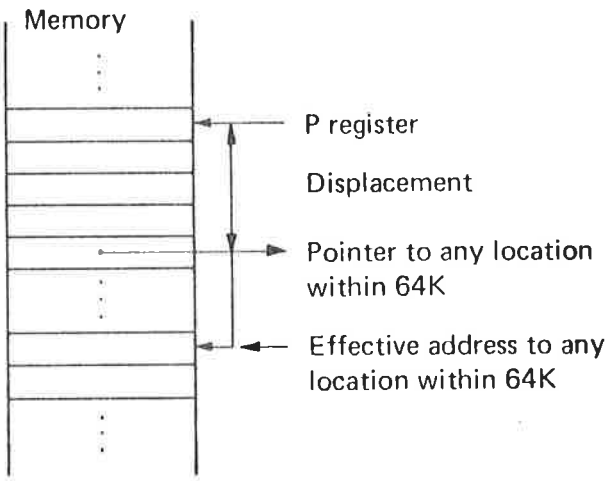


Figure 2.6: Schematic Illustration of Indirect P Relative Addressing

2.2.3 *B Relative Addressing*

The above two addressing modes are quite sufficient; in fact, theoretically, either one alone is sufficient. However, if the NORD-10 provided only one or both of the two addressing modes already described, it would not be particularly convenient or efficient to program. For instance, suppose that two sub-programs, each a couple of hundred words long, need to communicate. Within each sub-program memory accesses are commonly made using P relative addressing, or occasionally, indirect P relative addressing. But between the subprograms indirect P-relative addressing would have to be used almost exclusively since, in general, locations in one sub-program which instructions in the other sub-program must access will not be less than 128 words apart. But this is very inefficient since both subprograms contain indirect pointers to data and instructions local to the other sub-program.

To get around this inefficiency, another addressing mode is available, *B relative addressing,* which permits both sub-programs to directly address a common data area. B register relative addressing is specified by setting the X and I bits to zero and the B bit to one in memory address instructions. This addressing mode is quite closely related to P relative addressing, but instead of the displacement being added to the current value of the location counter, the displacement is added to the current value of the B register, and the resulting sum is used to specify the memory location accessed.

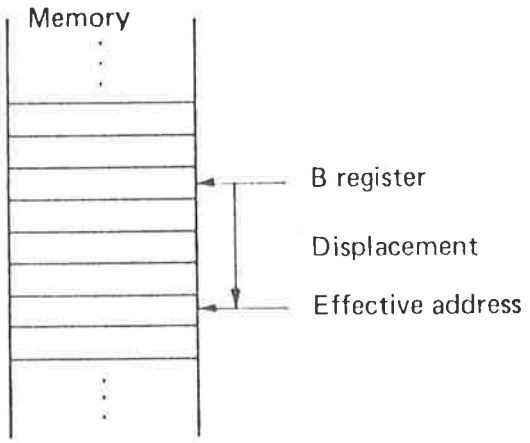


Figure 2.7: Schematic Illustration of B Relative Addressing

As an example, suppose location 405 contains the instruction LDA −4, B (044774_8) and the B register contains the value 10035 and the instruction in location 405 is executed causing the content of location 10031 to be loaded into the A register. The minus 4 in the displacement field of the LDA instruction in location 405 is added to the contents of the B register, 10035, giving a sum of 10031, and the contents of locations 10031 are loaded into the A register.

2.2.4 *Indirect B Relative Addressing*

Naturally, there is also an *indirect B relative* addressing mode which is specified by setting the B and I bits to one and the X bit to zero in memory address instructions, which has the same relationship to B relative addressing as indirect P relative addressing has to P relative addressing. This permits a subprogram to access data or locations in other sub-programs indirectly via pointers in an area common to several subprograms. This address mode may be used for calling library routines.

As an example, suppose location 10031 contains the instruction JPL I 3,B (135403_8) and the B register contains 400, a pointer to an area common to several sub-programs. Further suppose location 403 contains the value 2000. If the instruction in location 10031 is executed, the subroutine beinging at location 2000 will be called. The displacement, 3, in the JPL instruction is added to the contents of the B register, 400, giving a result of 403. The contents of location 403, 2000, are then used as a pointer to the subroutine.

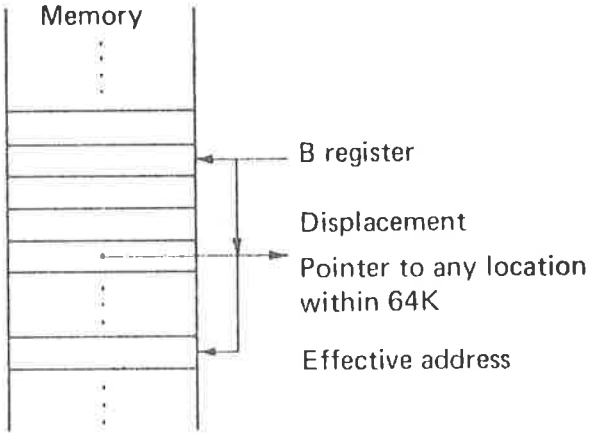


Figure 2.8: Schematic Illustration of Indirect B Relative Addressing

2.2.5 *X Relative (or indexed) Addressing*

The other four addressing modes all involve use of the X register: the simplest of these is X relative addressing which works like P and B relative addressing, but the displacement is added to the X register's content during the address calculation instead of to the content of the P or B register. This addressing mode is often used for randomly accessing the elements of a block of data.

For instance, suppose a recursive subroutine* upon being called saves the contents of the L, A and B registers in a three word block on a push down stack, and the X register points to the first free register in the stack. The following code might then be found at the beginning of the recursive subroutine:

* If you are unfamiliar with the concept of recursion, skip this example.

```
SUB,    STA  1,X
        COPY  SL  DA
        STA  2,X
        COPY  SB  DA
        STA  0,X
        AAX  3
        . . .
        . . .
        . . .
```

The effect of this code is illustrated in the following figure:

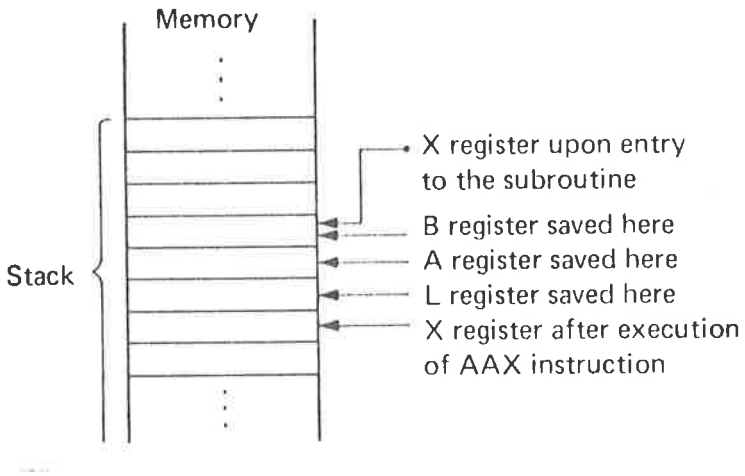


Figure 2.9: Example of Use of X Relative Addressing

For another example, re-read Section 2.2.3, **B Relative Addressing**, mentally substituting "X register" for "B register".

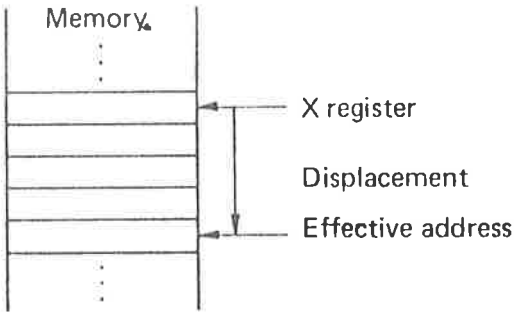


Figure 2.10: Schematic Illustration of X Relative Addressing

2.2.6 *B Relative Indexed Addressing*

The next addressing mode is called *B relative* indexed addressing; it is specified by setting the X and B bits to one and the I bit to zero in memory address instructions. In this mode the contents of the X and B registers and the displacement are all added together to form the effective address.

B relative indexed addressing is often very useful; for instance, when accessing row by row elements of a two dimensional array stored column by column. However, such uses tend to be difficult to describe, so we shall not attempt a description of one here.

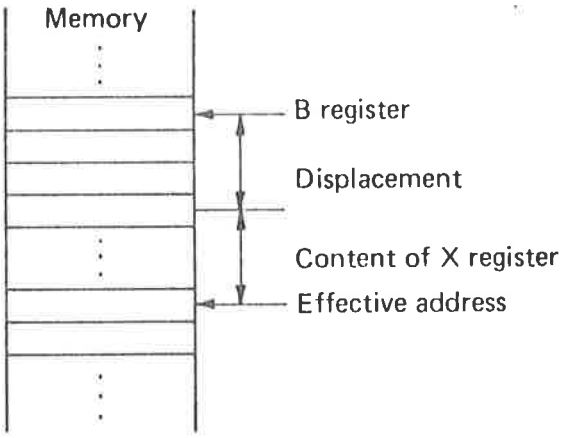


Figure 2.11: Schematic Illustration of B Relative Indexed Addressing

2.2.7 *Indirect P Relative Indexed Addressing*

The last two addressing modes are a little difficult to describe but very useful. *Indirect P relative indexed* addressing is selected by setting the X and I bits to one and the B bit to zero in the memory address instruction. This mode allows successive elements of an array based at an arbitrary place in memory to be accessed in a convenient manner.

The address calculation in the mode takes place as follows. The contents of the P register, say 4002, are added to the displacement, say −1, and produce a sum, 4001. The contents of the location 4001, say 10100, are added to the contents of the X register, say −100, to produce a new sum, 10000, the effective addresss. By incrementing the X register, successive locations may be accessed. For instance, using the above example, locations 10000 through 10100 can be successively accessed by stepping the contents of the X register from −100 to zero.

Readers are advised to go over this example carefully: stepping through an array in this fashion is done very often.

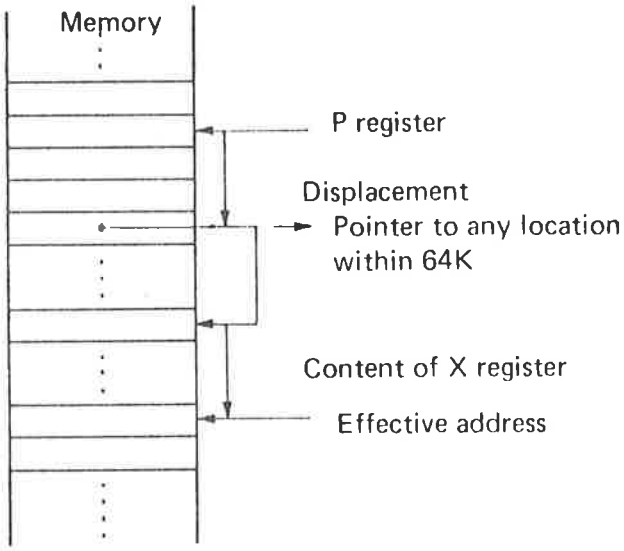


Figure 2.12: Schematic Illustration of Indirect P Relative Indexed Addressing

2.2.8 *Indirect B Relative Indexed Addressing*

The addressing mode, *indirect B relative indexed,* is identical to indirect P relative indexed addresssing except that the content of the B register is used in place of the content of the P register i the effective address computation. This mode can therefore be used to step through arrays pointed to from a data area common to several sub-programs.

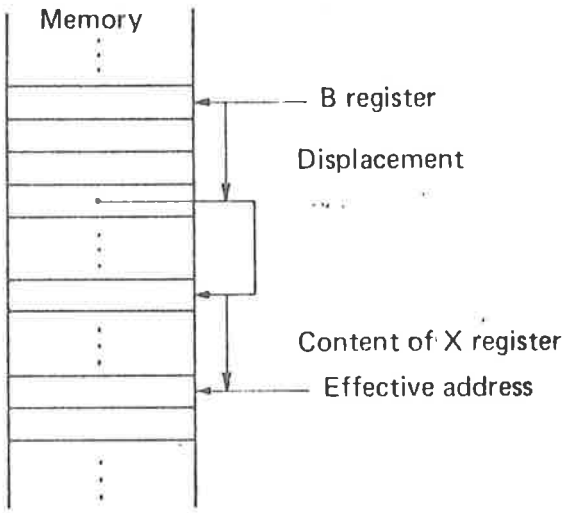


Figure 2.13: Schematic Illustration of Indirect B Relative Indexed Addressing

2.2.9 *Byte Addressing*

To facilitate the handling of character strings, the NORD-10 provides two instructions for byte handling, load byte, LBYT, and store byte, SBYT. Because of the requirement of full 64K addressing, the LBYT and SBYT use an addressing scheme different from the normal NORD-10 addressing.

For byte addressing, two of the NORD-10 registers, the T and X registers are used for addressing the byte. The contents of the T register point to the beginning of the character string, and the contents of the X register point to a byte within this string. Thus, the address of the word which contains the byte equals:

$$(T) + 1/2\,(X).$$

If the X register is even the byte is in the left part of the word, if X is odd, the byte is in the right part of the word.

A byte consists of eight bits.

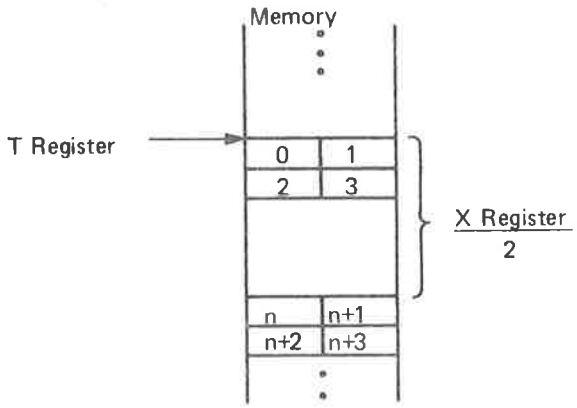


Figure 2.14: Schematic Illustration of Byte Addressing

2.2.10 *A Word about Nomenclature*

We have tried above to name the various addressing modes systematically in the hope that this would aid the reader in understanding the function of each mode and the modes' inter-relations. Unfortunately, the different addressing modes are generally referred to by much less precise names outside of this document: P relative addressing is often called normal addressing, indirect P relative is simply called indirect addressing, double indexing is a common synonym for indexed B relative addressing, indexed indirect is used for indirect P relative indexed addressing, etc. The X register is used in a manner commonly called post-indexing in the last two addressing modes. The B and P registers in all addressing modes are used in a manner commonly called pre-indexing.

2.2.11 *Summary of the NORD-10 Addressing Structure*

The addressing structure described above permits 1024 memory locations to be directly addressed at a given time (−128 to 127 relative to the contents of each of the P, B and X registers and the sum of the X and B registers). Any location in memory can be indirectly addressed.

The addressing modes are summarized in Appendix D.

2.3 *NORD-10 INSTRUCTION REPERTOIRE*

In the NORD-10 all instructions occupy a single word, 16 bits, yielding a very efficient use of memory, and also producing code with unusual efficiency, with regard to speed. 48 bits floating point arithmetic operations and floating integer conversions are standard. The optional 32 bits floating point format is described in Appendix E.

There are 8 general registers:

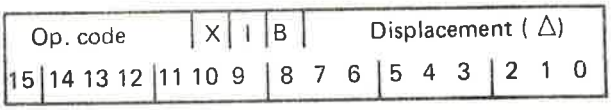
A	A register
D	D register
T	T register
L	L register
X	X register
B	B register
P	Program counter
STS	Status register containing TG, K, Z, Q, O, C, M indicators

and the following special registers:

OPR	Operator's panel switch register
LMP	Lamp register
PGS	Paging status register
PCR	Paging control register
PVL	Previous level register
IIC	Internal interrupt code
IIE	Internal interrupt enable
PID	Priority interrupt detect
PIE	Priority interrupt enable
ALD	Automatic load descriptor
PES	Memory error register
IR	Instruction register
PEA	Memory error address

In the following Δ is equal to the displacement, and EL is equal to effective address.

2.3.1 *Memory Reference Instructions*



Effective address:

	000000	Address relative to P	EL = P + Δ
,X	002000	Address relative to X	EL = X + Δ
I	001000	Indirect address	EL = (P + Δ)
,XI	003000	Post-indexing	EL = (P + Δ) + X
,B	000400	Address relative to B	EL = B + Δ
,X ,B	002400	Address relative to B and X	EL = B + Δ + X
I, B	001400	Pre-indexing	EL = (B + Δ)
,XI, B	003400	Pre- and post-indexing	EL = (B + Δ) + X

Store instructions:

STZ	000000	Store zero	(EL): = 0
STA	004000	Store A	(EL): = A
STT	010000	Store T	(EL): = T
STX	014000	Store X	(EL): = X
MIN	040000	Memory increment and skip next instruction if zero	(EL): = (EL) + 1

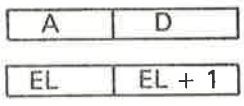
Load instructions:

LDA	044000	Load A	A: = (EL)
LDT	050000	Load T	T: = (EL)
LDX	054000	Load X	X: = (EL)

Arithmetical and logical instructions:

ADD	060000	Add to A (C, O and Q may also be affected)	A: = A + (EL)
SUB	064000	Subtract from A (C and Q may also be affected)	A: = A − (EL)
AND	070000	Logical AND to A	A: = A ∧ (EL)
ORA	074000	Logical inclusive OR to A	A: = A V (EL)
MPY	120000	Multiply integer (O and Q may also be affected)	A: = A * (EL)

Double word instructions:



STD	020000	Store double word	(DW): = AD
LDD	024000	Load double word	AD: = (DW)

Standard floating instructions:

TAD

T	A	D

FW

EL	EL + 1	EL + 2
Exponent		Mantissa

STR = STF	030000	Store floating accumulator	(FW): = TAD
LDR = LDF	034000	Load floating accumulator	TAD: = (FW)
FAD	100000	Add to floating accum. (C may also be affected)	TAD: = TAD + (FW)
FSB	104000	Subtract from floating accum. (C may also be affected)	TAD: = TAD − (FW)
FMU	110000	Multiply floating accum. (C may also be affected)	TAD: = TAD*(FW)
FDV	114000	Divide floating accum. (Z and C may also be affected)	TAD: = TAD/(FW)

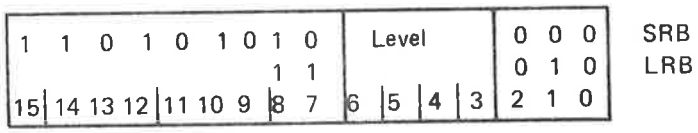
Byte instructions:

Addressing:

EL = (T) + (X)/2

Least significant bit of X = 1		Right byte
Least significant bit of X = 0		Left byte
SBYT	142600	Store byte
LBYT	142200	Load byte

2.3.2 *Register Block Instructions*



Addressing:

EL = (X) on current level

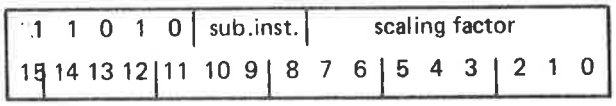
Register block instructions are privileged instructions.

LRB	152600	Load register block: P on spec. level: = (EL) X on spec. level: = (EL) + 1 T on spec. level: = (EL) + 2 A on spec. level: = (EL) + 3 D on spec. level: = (EL) + 4 L on spec. level: = (EL) + 5 STS on spec. level: = (EL) + 6 B on spec. level: = (EL) + 7
SRB	152402	Store register block

Specified level:

0	000000	Level 0
01	000010	Level 1
.		
.		
.		
017	000170	Level 15

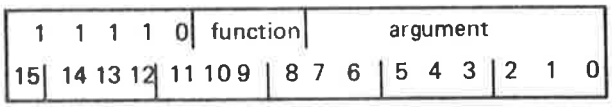
2.3.3 *Floating Conversion (Standard Format)*



NLZ	151400	Convert the no. in A to a floating no. in TAD
DNZ	152000	Convert the floating no. in TAD to a fixed point no. in A
NLZ+20	151420	Integer to floating conversion
DNZ−20	152360	Floating to integer conversion

The range of scaling factor is −128 to 127 which gives converting range from 10^{-39} to 10^{39}.

2.3.4 *Argument Instructions*

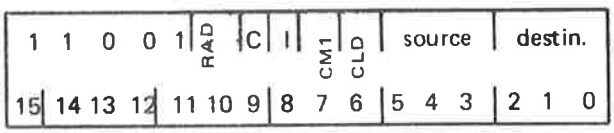


Function:

SAA	170400	Set argument to A	A: = ARG
AAA	172400	Add argument to A	A: = A + ARG
SAX	171400	Set argument to X	X: = ARG
AAX	173400	Add argument to X	X: = X + ARG
SAT	171000	Set argument to T	T: = ARG
AAT	173000	Add argument to T	T: = T + ARG
SAB	170000	Set argument to B	B: = ARG
AAB	172000	Add argument to B	B: = B + ARG

Argument is a signed number ranging from −128 to +127.

2.3.5 *Register Operations*



Arithmetic operations, RAD = 1:

C, O and Q may be affected by the following instructions:

RADD	146000	Add source to destination	(dr): = (dr) + (sr)
RSUB	146600	Subtract source from destination	(dr): = (dr) − (sr)
COPY	146100	Register transfer	(dr): = (sr)
AD1	000400	Also add one to destination	(dr): = (dr) + 1
ADC	001000	Also add old carry to destination	(dr): = (dr) + C

Logical operations, RAD = 0:

SWAP	144000	Register exchange	(sr): = (dr): (dr): = (sr)
RAND	144400	Logical AND to destination	(dr): = (dr) ∧ (sr)
REXO	145000	Logical exclusive OR	(dr): = (dr) ∀ (sr)
RORA	145400	Logical inclusive OR	(dr): = (dr) V (sr)
CLD	000100	Clear destination before operation	(dr) = 0
CM1	000200	Use one's complement of source	(sr) = $(sr)_o$
CM2	000600	Two's complement (CM1 AD1)	

Combined instructions:

EXIT	146142	= COPY SL DP	Return from subroutine
RCLR	146100	= COPY	Register clear
RINC	146400	= RADD AD1	Register increment
RCDR	146200	= RADD CM1	Register decrement

Specify source register (sr):

SD	000010	D register as source
SP	000020	P register as source
SB	000030	B register as source
SL	000040	L register as source
SA	000050	A register as source
ST	000060	T register as source
SX	000070	X register as source
	000000	Source value equals zero

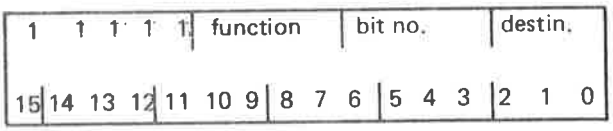
Specify destination register (dr):

DD	000001	D register as destination
DP	000002	P register as destination
DB	000003	B register as destination
DL	000004	L register as destination
DA	000005	A register as destination
DT	000006	T register as destination
DX	000007	X register as destination

Extended arithmetic operations:

RMPY	141200	Multiply source with destination; result in double accumulator	AD: = (sr) * (dr)
RDIV	141600	Divide double accumulator with source; quotient in A, remainder in D (AD = A * (sr) + D)	A: = AD//(sr)

2.3.6 *Bit Instructions*



BSKP	175000	Skip next location if specified condition is true	P: = P + 1
BSET	174000	Set specified bit equal to specified condition	
BSTA	176200	Store and clear K	(B): = K; K: = 0
BSTC	176000	Store complement and set K	(B): = K_o; K: = 1
BLDA	176600	Load K	K: = (B)
BLDC	17640	Load bit complement to K	K: = $(B)_o$
BANC	177000	Logical AND with bit compl.	K: = K ∧ $(B)_o$
BORC	177400	Logical OR with bit compl.	K: = K V $(B)_o$
BAND	177200	Logical AND to K	K: = K ∧ (B)
BORA	177600	Logical OR to K	K: = K V (B)

Specify condition:

ZRO	000000	Specified bit equals zero	(B): = 0
ONE	000200	Specified bit equals one	(B): = 1
BAC	000600	Specified bit equals K	(B): = K
BCM	000400	Complement specified bit	(B): = $(B)_o$

Specify bit number:

0	000000	Specifies bit in dest. reg.	B: = 0
0010	000010		B: = 1
0020	000020		B: = 2
.	.		.
.	.		.
0170	000170		B: = 15

For destination (D) mnemonics, refer to the section for register operations, 2.3.5. D = 0 specifies STS register.

Specify control flip-flop:

SSTG	000010	specifies floating rounding	B = TG
SSK	000020	specifies one bit accum.	B = K
SSZ	000030	specifies floating p. overflow	B = Z
SSQ	000040	specifies dynamic overflow	B = Q
SSO	000050	specifies static overflow	B = O
SSC	000060	specifies carry	B = C
SSM	000070	specifies multishift link	B = M

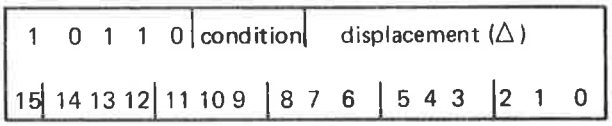
2.3.7 *Sequencing Instructions*

Unconditional jump:

For instruction word format and effective address, see Section 2.3.1 for memory reference instructions.

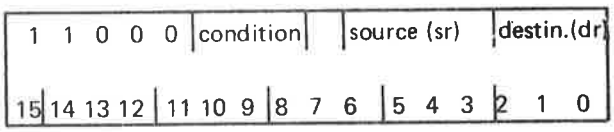
JMP	124000	Jump	P = EL
JPL	134000	Jump to subroutine	L = P; P = EL

Conditional jump:



JAP	130000	Jump if A is positive	P = P ± Δ if:A ≥ 0
JAN	130400	Jump if A is negative	P = P ± Δ if:A < 0
JAZ	131000	Jump if A is zero	P = P ± Δ if:A = 0
JAF	131400	Jump if A is non-zero	P = P ± Δ if:A ≠ 0
JXN	133400	Jump if X is negative	P = P ± Δ if:X < 0
JXZ	133000	Jump if X is zero	P = P ± Δ if:X = 0
JPC	132000	Increment X and jump if positive: X = X + 1	P = P ± Δ if:X ≥ 0
JNC	132400	Increment X and jump if negative; X = X + 1	P = P ± Δ if:X < 0

Skip instructions:



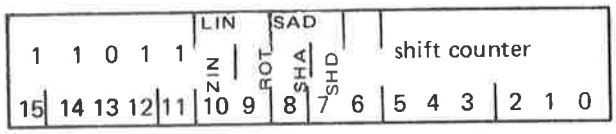
SKP	140000	Skip next instruction if specified condition is true	P = P + 1

Specified condition:

EQL	000000	Equal to
UEQ	002000	Unequal to
GRE	001000	Signed greater or equal to
LST	003000	Signed less than
MLST	003400	Magnitude less than
MGRE	001400	Magnitude greater or equal to
IF	000000	May be used freely to obtain
O	000000	easy readability

For source and destination mnemonics, see section for register operations, 2.3.5.

2.3.8 *Shift Instructions*

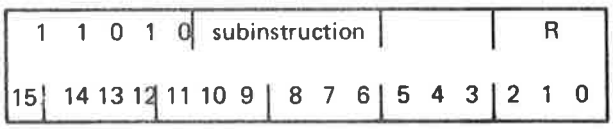


SHT	154000	Shift T register
SHD	154200	Shift D register
SHA	154400	Shift A register
SAD	154600	Shift A and D register connected
	000000	Arithmetic shift. During right shift, bit 15 is extended. During left shift, zeros are shifted in from right.
ROT	001000	Rotational shift. Most and least significant bits are connected.
ZIN	002000	Zero end input
LIN	003000	Link end input. The last vacated bit is fed to M after every shift instruction.
SHR		Shift right, gives negative shift counter. Note that SHR must precede the specified shift counter.

2.3.9 *Transfer Instructions*

Transfer instructions are privileged instructions.

Level independent instructions:



TRA	150000	Transfer specified register to A

Specified register R:

STS	1	Status register
OPR	2	Operator's panel switch register
PSR	3	Paging status register
PVL	4	Previous level code register
IIC	5	Internal interrupt code register
PID	6	Priority interrupt detect register
PIE	7	Priority enable detect register
ALD	12	Automatic load descriptor
PES	13	Parity error status register
PEA	15	Parity error address register

TRR	150100	Transfer A to

Specified register R:

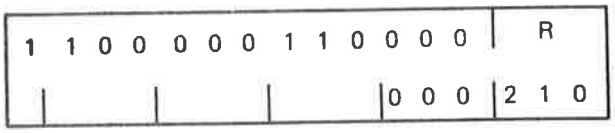
STS	01	Status register (bits 1-7)
LMP	02	Panel data display buffer register
PCR	03	Paging control register
IIE	05	Internal interrupt enable register
PID	06	Priority interrupt detect register
PIE	07	Priority interrupt enable register

MCL	150200	Masked clear of specified register
MST	150300	Masked set of specified register

Specified register:

STS	000001	Status register (bits 1-7)
PID	000006	Priority interrupt detect register
PIE	000007	Priority interrupt enable register

2.3.10 *Execute Instruction*

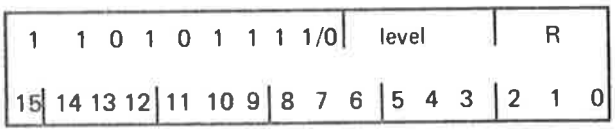


EXR	140600	Execute instruction found in specified register

Specified register R:

SD	000010	D register
SB	000030	B register
SL	000040	L register
SA	000050	A register
ST	000060	T register
SX	000070	X register

Inter-level instructions:



IRR	153600	Inter Register Read A: = specified register on specified level
IRW	153400	Inter Register Write Specified register on specified level: = A

IRR and IRW are privileged instructions.

Specified register R:

	000000	Status register
DD	000001	D register
DP	000002	P register
DB	000003	B register
DL	000004	L register
DA	000005	A register
DT	000006	T register
DX	000007	X register

Specified level:

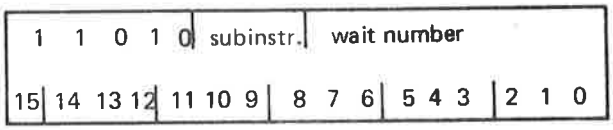
0000	000000	Level 0
0010	000010	Level 1
.	.	.
.	.	.
.	.	.
0170	000170	Level 15

2.3.11 *System Control Instructions*

System control instructions are privileged instructions.

ION	150402	Turn on interrupt system
PON	150410	Turn on paging system
IOF	150401	Turn off interrupt system
POF	150404	Turn off paging system

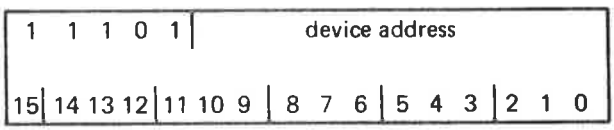
Halt instruction:



WAIT	151000	I	When interrupt system off: halts the program and enters the operator's communication.
		II	When interrupt system on: give up priority. If there are no interrupt requests on any level, the program on level zero is entered.

It is legal to specify a WAIT NUMBER 0 — 377_8.

2.3.12 *Input/Output Control*

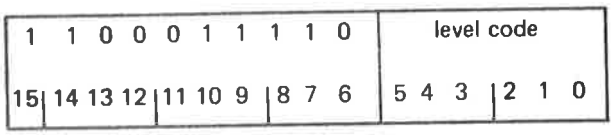


IOX 164000 Transfer data to/from specified device.

IOX is a privileged instruction.

NORD-10 may also be delivered with a NORD-1 compatible I/O instruction IOT.)

2.3.13 *Interrupt Identification*



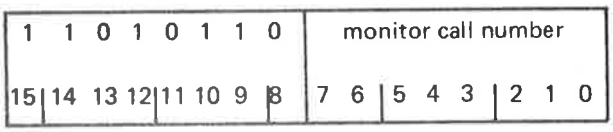
IDENT	143600	Transfer IDENT code of interrupting device with highest priority on the specified level to A register.

IDENT is a privileged instruction.

Level code:

PL10	000004	Level 10
PL11	000011	Level 11
PL12	000022	Level 12
PL13	000043	Level 13

2.3.14 *Monitor Calls*



MON	153000	The MON instruction is used in special different contexts when running under an operating system.

Examples:

MON	0	End of program or stop
MON	1	Read a character from specified device into the A register
MON	2	Output the character contained in the A register on the specified device.

3 BASIC MAC

3.1 *A SIMPLIFIED EXPLANATION*

3.1.1 *A Glance at the MAC Language*

A program in the MAC language, as in most other assembly languages, consists of a series of lines (records), each of which contains a command to the assembler or contains an instruction or constant which is to be assembled into a particular memory location. The particular memory location is selected by the value of an internal variable called the location counter. After each instruction or constant is assembled, the value of the location counter is increased by one. Thus, instructions and constants on successive lines are assembled into successive memory locations. Lines may have labels, a name (called a symbol) followed by a comma. Each label is associated with a value, the value of the location counter when the line containing the label is assembled. Labels provide a method of symbolically referencing a memory location. One command which may be given to MAC allows the user to assign an arbitrary value to the location counter. This command (called ORG in many assemblers) consists of a value followed by a slash. The following illustration may be helpful:

```
400/
L,          LDA     BUNNY
            JMP     L
BUNNY,      3
200/        3
```

The first line of the example sets the location counter to 400. The next line has a label, L, and indicates that the instruction LDA BUNNY is to be assembled into location 400. The instruction JMP L is to be assembled into location 401. Location 402 has a label, BUNNY, and the constant 3 is to be contained in that location. The next line sets the location counter to 200 and the constant 3 is to reside in this location.

The above should suffice as an introduction to the MAC language. The language is rigorously defined in Section 3.2.

3.1.2 *How MAC Works*

MAC utilizes two major tables as it processes the input stream (from any device). The symbol table contains an entry for each defined symbol including operation codes together with the symbols' values. As each new symbol is defined (usually as a label), the symbol and its value are entered in the table. The undefined symbol table contains an entry for each use of an undefined symbol. Every time an instruction or constant is assembled which contains a reference to a not yet defined symbol, the symbol and the location the instruction or constant is being assembled into are stored in the undefined symbol table. Later as undefined symbols become defined, this table is used to find memory locations which used the symbol when it was undefined and must now be updated.

Now study the flowchart on the next page.

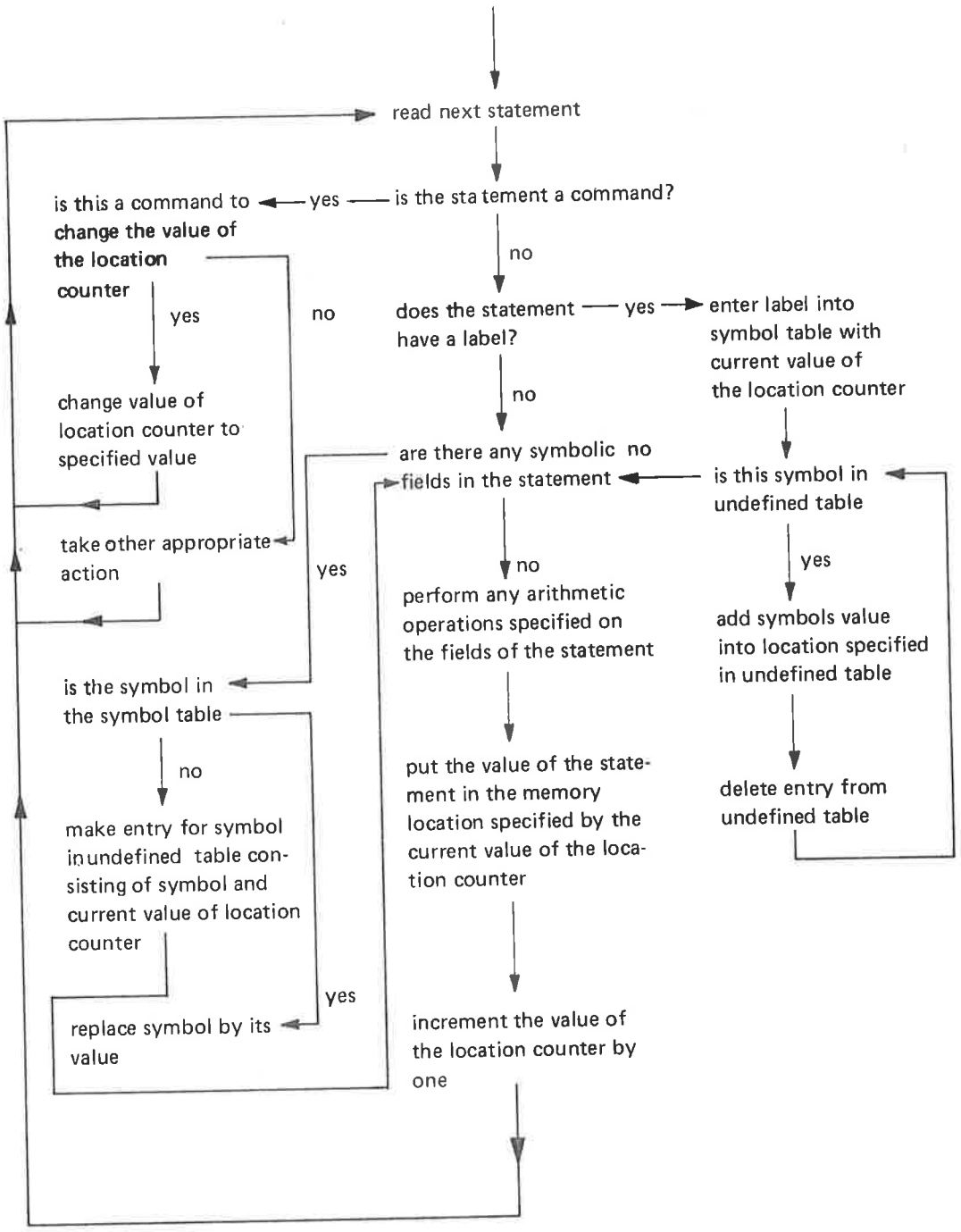


Figure 3.1: Simplified Flowchart of MAC

3.1.3 *MAC Input/Output*

MAC input/output is in some sense device independent. For example, MAC can accept input from the terminal, a file, or any peripheral device. All MAC's input/output is done via three logical data streams, each of which at any time is connected to files or devices. From now on we will let the word "file" mean any peripheral device or any collection of records on mass storage devices. These three logical data streams are called the source stream, list stream and object stream. All input to MAC comes from the file currently connected to the source stream. Symbolic output such as listings and error messages go to the file connected to the list stream, and binary output goes to the file connected to the object stream.

Later we shall see that these streams may be manipulated by commands. The command arguments are numbers or names recognized by the file management system.

This system, although it may be viewed as a separate system, is an integral part of the SINTRAN III, and adds powerful file management functions normally found on large computers.

A file is named with a character string, and this name is used in all commands to the file system. When a file is accessed, the file name must be connected to a file number and this number is used in the access routines.

Each file has one *owner* who has to be defined as a user in SINTRAN III. The owner is normally the user who created the file. A file is always allocated in the owner's area on the mass storage device (directory). Each user may declare up to eight other users as *friends* and give them privileged access possibilities to his files. Other users are regarded as *public users.*

The File Management System provides individual protection of files, with separate protection modes for the owner, owner's friends and the public users access of the file. A complete description is given in the manual "NORD File System" (ND-60.052).

Some combinations of assignments of files to streams are so commonly used, they have been given names. For instance, if the source stream is coming from the terminal, MAC is said to be in on-line mode; on the other hand, when the source is another file, MAC is said to be in off-line mode. The object stream (binary program) goes to memory during assembly and not to the connected file! This default condition is called absolute assembly mode, only being suspended by a BRF program sequence or BRF unit. This condition is often called BRF assembly mode beginning with)9BEG and ending with)9END.

The default file connected to the list stream is the dummy device (no. 0). The dummy device is generally used as a "bit sink"; for example, if no listing is desired, the listing is "printed" on the dummy device. However, error messages and other output generated by MAC are printed on the terminal. The interaction with MAC is somewhat dependent on the input/output system it is connected to. When running under SINTRAN III, MAC utilizes a special break strategy for terminal input. This allows the user to delete one character (A^C) or a sequence of characters (Q^C) back to the last break character. Break characters are carriage return and some others which have to be processed by MAC immediately, for instance, : and /.

Stand-alone systems do not offer the features mentioned above. In such systems note also the special interpretation of file names. Whenever a file name is parameter to a command it is looked up in the local symbol table. The symbol value is then used as a device number.

3.1.4 *MAC as a Debugging Aid*

Once a program is assembled into memory or loaded, MAC may be used to inspect and change the program, cause the program to start execution, add additional instructions or data to the program, and to perform many other functions useful for debugging. Such actions are inititated by a series of MAC commands typed from the terminal. These debugging directives are from the same set of MAC commands which, when found on a symbolic input file, control the function of the assembler. Some commands in this set are oriented more towards what is commonly thought of as assembly functions and also towards what is more commonly thought of as debugging functions; but, in fact, there is only one set of commands, and any may be used via any input file.

3.2 *DETAILED DESCRIPTION OF BASIC MAC*

3.2.1 *Basic Elements of MAC*

3.2.1.1 Characters

The most basic element in the MAC language is a character. All more complex elements are formed from characters. The character set used by MAC is 7-bit ASCII, right-justified in an 8-bit field with the left bit set to zero (see also Appendix D.4). The Operating System performs appropriate character conversions when communicating with input/output devices using other character sets. For instance, the terminals supplied by Norsk Data use 7-bit ASCII characters right justified in 8-bit fields with the eighth bit used to give the character even parity.

Some characters have a special meaning in the MAC language, either as commands or arithmetic operators or special symbols. Letters and digits are generally used to construct more complex linguistic elements. All characters may in some cases stand for themselves.

3.2.1.2 Numbers

There are three kinds of numbers in the MAC language: octal numbers, decimal numbers and floating point numbers. The latter two types of numbers will be described in Chapter 4 in "Extended MAC". Octal numbers are formed from the digits 0 through 7. When printed by MAC, octal numbers always have six digits and negative numbers are printed in two's complement notation. Octal numbers read by MAC consist of one or more digits in two's complement notation. The last six digits specify the number.

For Example:

INPUT	becomes	INTERNAL
3		000003
123456		123456
12600012		000012
777777		177777

3.2.1.3 Symbols

Symbols usually consist of a string of letters and digits including at least one letter. The following are legal symbols:

A
A3
B9D3
345A
93A234
Q1122443

Any number of letters and digits may be used in a symbol, but only the *last five* characters distinguish symbols. Thus, K12345 and L12345 are treated as the same symbol. Normally (in octal mode), the digits 8 and 9 are considered to be letters for the purpose of symbol constructions. Thus, 932 is a symbol, not a number. No user defined symbol can be defined without a letter or use a character other than a letter or number, but a number of special symbols built into MAC contain other characters. Every symbol has either a numerical value and is said to be *defined* or does not have a value and is said to be *undefined*.

There are several special symbols defined in MAC. These are listed below with an explanation of what their values are:

*	value is current value of location counter.
↑	value is content of memory location currently pointed to by location counter.
,B	value is 000400, B-relative bit in memory address instructions.
,X	value is 001000, index bit in memory address instructions.
# CC	value is a 16 bit number, the internal code for the two characters following the sharp sign (#).
# #A	value is a 16 bit number, the internal code for the character following the two sharp signs. The value is right-justified, and the most significant bits are zeroed.
# # #ABCD#	value is a 32 bit number. Six bits are used from the internal representation of the characters in the string enclosed by the third and the fourth sharp sign. The number is right-justified in a double word, and the most significant bits are zeroed if the string consists of less than five characters. If more than five characters, only the right-most five are used. The two most significant bits are always zero.

3.2.1.4 Expressions

Expressions consist of numbers and symbols separated or preceded by the arithmetic operators + and —. The shift operator, @ is *not* unary. In fact, when starting a line it acts as an abbreviation for the)LINE command which stops assembly.

```
200
A+B—3
260—200
1 @ 10 + B
```

Expressions are evaluated from left to right and use of parentheses is not allowed. The value of an expresion is the arithmetic sum of the values of the symbols and numbers. If A has the value 100 and B the value 3, the values of the four expressions alone are:

```
200
100
60
403
```

Other operations which are implicit are sometimes used when forming the value of an expression. These will be discussed in Section 3.2.2.4 - Instructions.

3.2.2 *Types of Statements*

We now begin a detailed explanation of the construction of the MAC language. For the purpose of this explanation it is convenient to divide these constructs into four categories: instructions, constants, commands and comments. Instructions and constants are used to represent the content of NORD-10 memory locations, commands instruct MAC itself to take various actions, and comments are used to clarify programs written in the MAC language and are ignored by MAC. A program in the MAC language consists of a series of lines, each terminated by a carriage return. (Line feeds are ignored by MAC except in strings. See also Section 3.2.3.7. For readability the carriage return of each input line should be followed by a line feed.) Each line consists of zero or more instructions, constants, commands and comments. A line consisting only of a carriage return is ignored by MAC.

3.2.2.1 Comments

A comment is introduced by the character % (percent sign) and continues to the end of a line. As already stated a comment is ignored by MAC. Any characters may occur within a comment except, of course, a carriage return which would end the comment. Here is an example:

```
% THIS IS A COMMENT
```

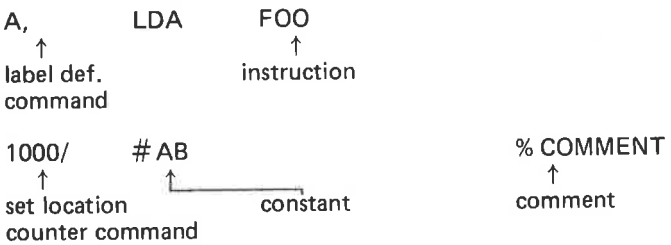
While it can stand alone on a line, a comment often shares a line with instructions, constants and commands; although when a command does not stand alone it, by definition, must come last on the line.

3.2.2.2 Commands

Commands take a variety of forms which will be described in detail below, but generally they appear first on a line and are perhaps followed by a comment. Some examples of commands are:

```
)FILL
*:
?
200!
```

Two commands commonly share a line with instruction and constants as well as comments. These are the label-definition commands and the set-location-counter commands. Again, these commands will be described in detail below, although examples follow immediately.



3.2.2.3 Introduction to Instructions and Constants

Instructions and constants usually appear one to a line, perhaps preceded by a set location-counter or label-definition command and perhaps followed by a comment. Each instruction represents exactly one memory word as do most constants. However, a few constants represent two or more memory words. Occasionally it is useful to put more than one instruction or constant on a line. This may be done by separating the several instructions and constants by semi-colons. Some examples follow:

```
        LDA     PER
          ↑
      instruction

        LDA     FOO                     % LOAD INSTRUCTION
          ↑                               ↑
      instruction                       comment

L,      01372
  ↑       ↑
command constant

        'ABCDEFG'                       % A STRING CONSTANT
           ↑                              ↑
        constant                        comment

        LDA     FOO;    STA PER
          ↑               ↑
      instruction     instruction

200/    LDA     FOO;    012345; STA PER     % ZOWIE
  ↑       ↑               ↑       ↑           ↑
command instruction     constant instruction comment
```

We now describe instructions, constants and commands in greater detail. No further discussion of comments is necessary.

3.2.2.4. Instructions

An instruction consists of a symbolic operation code and zero or more other symbols or numbers. These symbols or numbers are combined together using standard arithmetic addition and subtraction operations and a couple of rather obscure non-arithmetic operations indicated implicitly by the particular operation code. The addition and subtraction operations are represented by + (addition), space (addition), tab (addition) and — (subtraction).

A conventional and very readable format starts instruction eight character positions from the beginning of a line. Character position 13 is always left blank, as is position 12 unless the symbol I is included in the instruction and as is position 11 unless the op-code has four letters as in SWAP.

Beginning with character position 14, the rest of the elements of the instruction are found with successive elements separated by a single space, a plus sign or a minus sign; exceptions are the address mode specifies ,X and ,B which come last in the instruction and are separated from the preceding elements only by these commas. If the instruction is followed by a comment, readability is increased if all comments begin at a certain character position.

A tab, if available, is the easiest way to begin the comment in a certain column as well as the easiest way to skip over the fixed number of columns before beginning the instruction.

```
START,  LDD     11,X
        SWAP    SA DA
        STT     7,B
        LDA  I  FOO,B,X
        STA     FOO+3,X
        STX     ZOO—2+MUM
        JMP     *+4
```

We now delve a little deeper into the rules for combining the elements of an instruction. Let us suppose that in memory location 402 of the NORD-10 there resides an instruction to load the A register with the content of location 405. One instruction which has the desired function is 044003. This instruction may be thought to be made up in the following fashion:

044000 + 000003 + 000000

044000 = operation code for the load A register instruction
000003 = displacement of 3
000000 = addressing mode

That is, as the sum of the operation code, displacement, and addressing mode (if this is not clear, reread Section 2.1). Another instruction is 154407, which specifies that the A register should be arithmetically shifted left 7 bit positions. This instruction may also be thought of as sum.

154400 + 000000 + 000007

154400 = operation code
000000 = type of shift
000007 = direction and length of shift

Leaving out elements which add nothing to the sum and using space for the addition operator, these instructions might alternatively be written:

```
044000    3
154400    7
```

or hopefully in the more symbolic format

```
LDA  3
SHA  7
```

Unfortunately, it is not sufficient to merely think of each instruction as the sum of the values of the elements of the instruction. We run into trouble with the addressing structure of the computer.

What happens when, instead of writing

```
LDA  3
```

the user wants to write

```
LDA  FOO
```

where FOO is the label on the third memory location following the location containing the LDA instruction. Let us assume momentarily that the LDA instruction is in location 1000. Then FOO would have the value 1003. But

```
LDA 1003
```

is certainly different from

```
LDA  3
```

How can we treat all instructions the same, both those in which the user desires relative addressing and those in which the user wishes to address an absolute memory location? The solution lies in using the symbol * when relative addressing is desired. Thus,

```
LDA  3
```

becomes

```
LDA * +3
```

which, if * has the value 1000, is the same as

```
LDA  1003
```

But there is still a problem since we cannot just add the value of LDA which is 044000 to 1003 and get the instruction we desire, 044003. The answer is that if we now subtract the value of * from either representation, we get what we want. That is,

```
LDA * +3–*
```

is then the same as

```
LDA  FOO–*
```

in the case where FOO has the value 1003. MAC does this: it automatically subtracts the value of * from such instructions.

But there are still problems, as we shall see if we consider the instruction

```
JMP *–1
```

If we think of this as an arithmetic sum, things do not work out right. The value of the symbol JMP is 124000, the value of the symbol * is 010001 (the current value of the location counter) and the value of 1 is 1. But

```
124000 + 010001 – 000001 – 010001

124000 = JMP
010001 = *
–000001 = –1
–010001 = –current value of location counter
```

is 123777 instead of 124377 which the instruction should compile into. By using only 16 bit arithmetic operations we manage to change the operation code and have the wrong displacement. The problem in this case is that we wish to reference location –1 relative to the current location rather than absolute memory location 10000 which, when added in, causes the op. code to change. Even when MAC subtracts the current value of the location counter from the sum of the instructions, the result is 123777, which, although the displacement is correct, yields an incorrect operation code and the addressing mode. The problem is now that the borrow in the subtract 1 (one) operation carries out of the 8 bit displacement field. Some more complex "correction" is clearly necessary. What is more, a different correction is necessary for different hardware instruction formats since the width and position of subfields varies from format to format. We discuss this later.

In order to make it as easy as possible to write programs for the NORD-10 computer in the MAC language, MAC automatically computes the correct relative address, and the programmer writes all instructions as if fixed addresses were used. However, the programmer must not exceed the range of the relative address. For example, the instructions

```
STA   *–1
STT   *–3
LDA   *+4
ADD   *+5
STA   *+3
EXIT
```

are equivalent to writing

```
004377
010375
044004
060005
004003
146142
```

MAC distinguishes memory reference instruction and other kinds of instructions and takes the appropriate action in each case. Of course, for this to be possible, the user must use the predefined symbolic operation codes and addressing mode indicators.

For all memory address instructions the assembly of the instructions takes place in the following manner. First, the values of the symbolic operation code and address mode specifiers are added together - call this result "sum". Next, the values of the rest of the elements of the instruction are added or subtracted from sum. Then three values are possibly subtracted from sum, the current value of the symbol * (the location counter), the current value of the B location counter or the current value of the X location counter, depending on the address specification and according to the following chart.

	* location counter
,B	B location counter
,X	X location counter
	* location counter
,B	B location counter
,B,X	B location counter and X location counter
,X	* location counter
,B,X	B location counter

Now bit 7 of the sum is examined. If it is a 1, 400_8 is added to the sum, and this sum is now the completely assembled instruction. To the reader who is confused about the B location counter and the X location counter - relax! As opposed to the * location counter, which is automatically updated by MAC, the B and X location counters are static. The initial values are zero thus having no effect as described above unless they are affected by the user through the)9SET (in Section 3.2.3.9).

We will present some examples to show what this all means. Suppose we have the instruction LDA *+2. The value of the symbol LDA is 044000. To this is added the current value of * and the value of 2 giving, 044000 + the value of *+2. Since the specified addressing mode is P relative, the current value of * is next subtracted from the sum, leaving a value of 044002. Bit seven of this is zero, so assembly of the instruction is complete. We have the correct operation code, 044000, and positive displacement of 2 relative to the P register.

Next, suppose we have the instruction LDA *−2. To the value of LDA is added the current value of * and 2 is then subtracted from the sum. Since the addressing mode is P relative, the value of * is next subtracted giving

044000 + * − 2 − * = 044000 − 2 = 043776

Bit 7 is now examined and it is 1, so 400 is added to 043776 giving 044376 which is the correct instruction. The operation code is 044000 and the displacement is 376, minus 2 when considered to be a 7 bit two's complement number.

Should one of the symbols in an instruction be undefined (i.e., has no value) when the instruction is assembled, the rest of the instruction assembly takes place normally, and the value of the undefined symbol is later added or subtracted (whichever is appropriate) when the symbol becomes defined. For example, suppose the instruction LDA FOO resides in location 200, but when this instruction is initially assembled, the symbol FOO has not yet been defined. it cannot be added in. The addressing mode is P relative, so the current value of * is subtracted giving 043600 which is stored in location 200 and a note is made that when FOO becomes defined its value should be added to the value in location 200. Later FOO is defined to have the value 204 so that it is added to the value in 200 giving 044004, which is the correct instruction. Bit 7 is zero so 400 need not be added. If FOO is later defined to have the value 176, then 176 will be added to 043600, the content of location 200, giving 043776. Because FOO is defined as 176 but referenced from location 200, a negative displacement is assumed and 400 is added giving 044376, which correctly addresses the location two before location 200.

Note: Undefined symbol(s) in an instruction at the time the instruction is first assembled, may result in incorrect assembly when the instruction is later assembled. This is true when an argument or displacement (not P relative) turns out to be negative.

Symbolic Code:	First Assembly:	Later Assembly:	
SAA ARG	170400	170377	
LDA ,B A−B	044400	044401	044377
ARG = −1			
A = 1			
B = 2			

Observe that 400 has to be added to obtain correct instruction code in both cases. Such problems are avoided by using the two-pass assembly option.

Further examples of the use of instructions will be found in Chapter 5.

3.2.2.5 Constants

Constants are identical to instructions except for not having an operation code as the first element of the expression.

A + B + 300
* + 2
0 + LDA

Consequently, when a constant expression is evaluated, only the explicit arithmetic operations are performed. In other words, constants allow construction of arbitrary 16-bit numbers. As with instructions, the location counter is increased by one after each constant is assembled and stored in a memory location.

3.2.3 *The Commands in Basic MAC*

Commands have a number of different formats. For the most part, commands direct MAC to take some action and cause no instructions to be assembled, but there are exceptions.

Two commands are of paramount importance; the set location counter command and the define label command, which have briefly been mentioned. In this section these two commands and all the commands available in basic MAC will be described in detail. Commands available only as parts of options in extended MAC will be described under the appropriate option in Chapter 4.

3.2.3.1 Set-Location-Counter

This command is executed by writing an expression followed by a slash (/) at the beginning of a line. The expression is evaluated algebraically and may contain symbolic or numeric elements. However, all symbols used must have values. The location counter is then set to the value of this expression. Thus,

400/

sets the value of the location counter to 400. If the symbol A has the value of 600,

A/

sets the location counter to have the value 600.

A+3/

would set the location counter to 603. Execution of this command also has the side effect of printing out the content of the memory location now specified by the location counter if MAC is the on-line mode. This feature is used to examine memory locations from the terminal.

3.2.3.2 Define-Label

This command is executed by writing a symbol at the beginning of a line followed by a comma (,). When this command is executed, the specified symbol is given as its value, the current value of the location counter. Thus,

```
400/
A,
```

gives A the value 400. The comma in a label definition must not be confused with other uses of the comma, as for instance in the symbol ,X. There are some constraints on the definition of labels, but these will be discussed later.

With the above two commands, instructions and constants, programs can be written. For example:

```
400/    LDA     FOO
        STA     L
        JMP     *
FOO,    3
L,      0
```

This is equivalent to

```
location 400    044003
                004003
                124000
                000003
                000000
```

3.2.3.3 =

This = command is another method of giving a value to a symbol. The way to use this command is to write a symbol at the beginning of a line and to *immediately follow* the symbol by the = sign (no intervening characters including spaces). The = sign is then followed by an expression composed of symbols and numbers. The arithmetic value of this expression is made the value of the symbol

A=3

There may be no undefined symbols in the expression. This gives A a value of 3. The following programs are equivalent:

```
PER=4752
PER/      SHT     3
          JMP     *
```

and

```
4752/
PER,      SHT     3
          JMP     *
```

and

```
4752/
PER=*;    SHT     3; JMP *
```

During the definition of symbols using either , (comma) or = (equals), two rules should be followed:

1. A symbol which already has a value should not *generally* be redefined (even to the same value), but see Section 3.2.3.9 for an exception to this rule.

2. A symbol may not be given an undefined value.

The second rule is enforced by MAC which takes no action if the expression following an = sign contains an undefined symbol and will not permit the location counter to become undefined (via a set-location-counter command with an undefined element in the expression preceding the /). The first rule is not enforced by MAC except by a warning message. For each additional definition of a symbol a new value for the symbol is saved, and this new value is then used in all cases where the value of the symbol is needed. It is possible to delete the new value and get back to the previous value as is described in Section 3.2.3.9 in the)KILL command. Also discussed in Section 3.2.3.9 is the use of the = and)KILL command in conjunction, to change the value of a symbol and allow use of "local" and "global" symbols.

3.2.3.4 :

This command causes the value of the symbol to the left of the colon to be printed out on the terminal. The command can be used in two ways: if the colon immediately follows a symbol, the value of the symbol is printed, otherwise, the *arithmetic value* of the preceding expression is printed.

```
A=4
A:000004
3+A:000004
A+3:000007
A+3   :000007
3+A   :000007
```

If the colon is immediately preceded by a symbol and the symbol is undefined, the letter U will be printed instead of a value. If, however, a symbol does not immediately precede the colon and some symbol in the expression is undefined, when the symbol later becomes defined the value of the now defined symbol will be added to the content of the memory location to which the location counter was pointing when the colon command was executed. It is improbable that the user will desire this.

3.2.3.5 !

This command starts execution of a program at the location specified by the expression preceding the ! If the breakpoint option is not added, registers other than the P register are undefined. Further use of this command is discussed in the section describing the breakpoint option - Section 4.2.4.

3.2.3.6 <

This command sets up an interval used by other commands. The expression preceding the < is set as the lower bound of the interval and the expression following the < is set as the upper bound of the interval. For example:

```
300 < 4000
```

sets up an interval of 300 to 4000. If A has the value of 200,

```
A+300 < 4000
```

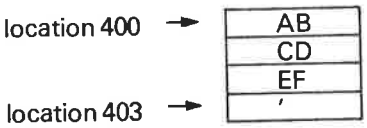
sets up an interval of 500 to 4000. If there is an undefined expression in one of the expressions, the command is ignored.

3.2.3.7 '

This command provides a method of storing an ASCII text string in memory. The format of the command is a single quote followed by a text string and terminated by another single quote. The characters of the text string are placed in successive memory words, two characters per word, starting at the location indicated by the current value of the location counter. The terminating single quote is considered to be part of the text string. After this command, the location counter points to the location one after the last location holding characters of the string. For example, if the string command

'ABCDEF'

is given and the location counter is 400, memory locations 400 through 403 will have the following contents after execution of this command and the location counter will have the value 404.



No provision has been made for including a single quote within a text string.

3.2.3.8 $

The user may determine which files are connected with the source, list and object streams using the $ command (see Section 3.1.3). The $ command has three octal arguments which precede the $. For example:

2,5,3$

Each argument is a file number. The first argument determines the files connected to the source stream, the second determines the file connected to the list stream, and the third determines the file connected to the object stream.

If less than three arguments are supplied, the file assignments to only some of the logical streams are changed. For example,

2,0$

changes only the source and list stream device assignments, and

2$

changes only the source assignment.

However, if no argument to the $ command is furnished (i.e., $ is given alone), the file specifications before the last)LINE command are used as default.

Users are recommended to use the)9ASSM command (described later) which also accepts file names.

3.2.3.9)

This command starts program execution at a location specified by the value of the symbol immediately following the). For example,

)SYMBOL

causes a jump to the address given by the value of the symbol SYMBOL. The symbol must be defined.

The following are technically not commands but library routines. However, these library routines are used like commands so they will be described in this section. These routines are all called using the) command. After a routine has performed its job, it jumps back to the correct place in MAC to continue processing the input stream. We shall call these routines commands hereafter.

)9MSG

The text following the)9MSG command up to and including the next carriage return is printed on the device associated with the list stream. For example,

)9MSG THIS MESSAGE IS PRINTED

causes the text

THIS MESSAGE IS PRINTED

to be printed on the list file. If the list file is set to zero (dummy device), the message will appear on the terminal.

)BPUN

This command outputs to the object stream a binary dump of the area of memory specified by the < command. The file has a leader and a loader on the front and may be read into the NORD-10 by the hardware loader, the operating system or by MAC itself ()9READ). The format is often called absolute binary format as opposed to binary relocatable format. A checksum is generated and checked by the load program. One or two symbols must be specified after the)BPUN command (not expressions).

)BPUN PER BOOTE

When the binary dump is read back into the NORD-10 the program may be started at the location specified by the value of PER at the time the file was generated. The value of the second symbol determines the end location of the loader. If this argument is not supplied, the loader will be located as in current memory.

)CLEAR

The command clears MAC's tables so that another program may be assembled without confusion due to doubly defined symbols. The)CLEAR command also "zeros" the B and X location counters.

)FILL (and literals)

For those users who understand literals,)FILL dumps the literal table, and literals are preceded by a left parenthesis (( ). For users who are unfamiliar with literals, we shall first describe them.

Very often one will want to use and refer to a constant in a program. For instance,

```
        LDA     FOO
        ADD     TWO
        STA     FOO
        JMP     *
TWO,    2
```

TWO is a reference to the constant 2. If one uses many constants, it can become quite tedious to actually write and label all of the constants.

```
        LDA     ZEROO
        ADD     ONE
        ADD     TWO
        ADD     THREE
        ADD     FOUR
        STA     SUM
        JMP     *
ZEROO,  0
ONE,    1
TWO,    2
THREE,  3
FOUR,   4
SUM,    0
```

Literals were invented to save some of this effort. Using literals the above would be rewritten:

```
        LDA     (0
        ADD     (1
        ADD     (2
        ADD     (3
        ADD     (4
        STA     SUM
        JMP     *
SUM,    0
```

The (is called the literal marker. The idea is that the assembler will now set aside five memory locations somewhere containing the values 0, 1, 2, 3 and 4 and will also treat (0, (1, (2, (3, and (4 like ligitimate symbols whose values are the locations holding the 0 through 4. To try to make it clearer, it is just as if the user had written:

```
SOMEWHERE/
(0,         0
(1,         1
(2,         2
(3,         3
(4,         4
```

A literal causes a value to be placed somewhere with a label which is the literal itself. But now the problem of locating "somewhere" arises. The user wants some control of where the assembler puts these values since the assembler must not just randomly place them, perhaps over other programs. Thus, we have the)FILL command. As MAC processes literals it does not immediately place the values in memory locations but instead saves the values up in an internal table along with the location referencing the literal. Later, when the)FILL command is given, MAC dumps all of the literal values it has collected into the location pointed to by the location counter and successive locations. Although the user can control *where* the literal values are placed, the order of the literal values is not defined. Thus, if the user writes,

```
400/        LDA     (0
            ADD     (3
500/
)FILL
```

locations 500 and 501 will contain 0 and 3 in some order, and the LDA and ADD instructions will refer to the appropriate locations. Obviously, the literal values must be able to be addressed from the instructions that access them, i.e., within 128 locations of the accessing instruction.

There are some limitations in the MAC implementation of literals:

1. Nested literals are not allowed, e.g. ((3.

2. Even if nested literals were allowed, some expressions would not be possible since there is no "close literal" mark.

3. In general, literals with multiple location values are not allowed, with the exception of floating point numbers.

The MAC implementation is reasonable in some ways. For instance, the following two programs are equivalent.

```
            LDA     (4-N
            COPY    SA DD
            LDT     (141
)FILL
N=2
```

```
            LDA     (2
            COPY    SA DD
            LDT     A
)FILL
A,          141
```

The)FILL command may be abbreviated by &.

)KILL

This command is used to expunge symbols from the symbol table or undefined symbol table. The command is useful when the symbol table becomes overfull, causing the assembler to run slowly or preventing further definition of symbols. The)KILL command is also used in conjunction with the = command to change the value of a symbol. For instance,

```
A=3
B=A
)KILL  A
A=B+1
)KILL  B
```

adds one to the value of A.

Combined use of)KILL and = also allows "declaration" of "local" symbols. For example,

```
A = 3
...
...
...
A = 2
...
...
...
)KILL  A
...
...
...
```

In the interval between the A = 2 command and the)KILL A command, A has a value of 2. Outside that interval A has a value of 3.

Following the)KILL command there may be as many symbols (separated by spaces) as can fit on a line. An attempt is made to delete each symbol in the list. For each symbol, the symbol table is first searched and the last definition of the symbol deleted if any definition of the symbol can be found. If none is found, the undefined symbol table is searched and the last instance of this symbol is deleted. If no instance of this symbol is found in either table, MAC gives up and goes on to the next symbol in the list following)KILL. For each symbol deleted MAC prints a plus sign (+) on the terminal if MAC is in on-line mode. Note that multiple definitions of a symbol in combination with the)KILL command enable a symbol name to be used as a push down stack.

Single symbols may be deleted with the ← command which has the form

SYMBOL←

which is identical to the command

)KILL SYMBOL

)LINE

The)LINE command switches the source stream to standard input (terminal), and the list stream to the dummy device (number 0). However, the files are not closed, but the current stream specifications are saved. These specifications are used if a)9ASSM or $ command, without arguments, is encountered. Note that)9ASSM or $ *without arguments* from an input file other than standard input will have no effect whatsoever! The)LINE command may be abbreviated by @.

Each time the)LINE command is executed during the assembly, the number of errors or warnings issued since the last)LINE (or start) is output on the terminal like this:

**** &&&&&& DIAGNOSTICS ****

where &&&&&& is a six digit octal number. This information is given even if the error count is zero.

)NWRT

The command puts MAC in non-write mode. That is,)WRITE commands are ignored. The)WRTM command puts MAC back to write mode. MAC is initially in non-write mode.

)PRINT

The)PRINT command causes the contents of memory in the interval last specified by the < command to be printed out on the list stream device.

)PUNCH

The)PUNCH command produces output similar to that of)PRINT, but the output goes to the file associated with the object stream. The format which is often called octal dump is suitable for loading using the NORD-10's automatic read mode.

)9SET

The command is used to set the value of the B and X location counters. The command has two arguments, the first of which must be one of the symbols ,X and ,B and the second must be a defined symbol, not an expression. The arguments are delimited by spaces. For example,

```
A=400
)9SET     ,B  A
```

sets the value of the B location counter to 400 while

```
B=A+1000
)9SET     ,X  B
```

sets the value of the X location counter to 1400.

People writing NORD-10 programs using previous versions of MAC quite commonly set up a "common storage area" with the B register pointing to the middle of the area. For example,

```
% COMMON STORAGE AREA
          ...
          ...
A1,       0
G,        0          % center of common storage area
BR2,      0
...
...
% PROGRAM START
START,    LDA        (G
          COPY       SA DB
          ...
          ...
```

Once this is done, individual locations in the common storage area are accessed as follows:

```
          ...
          LDA        A1-G,B
          STA        BR2-G,B
          ...
```

MAC itself references symbols common to more than one routine this way. However, it can grow tiring writing the "-G" all the time. Hence, the)9SET command. If the program above at START were changed to

```
% PROGRAM START
)9SET     ,B  G
START,    LDA        (G
          COPY       SA DB
          ...
```

then the locations in the common storage area could simply be accessed as follows:

```
...
LDA       A1,B
STA       BR2,B
```

Of course, the)9SET command only has an affect at assembly time, and it is the programmer's responsibility to make sure that at run-time the B or X register is set up in a manner which makes sense of the assembly time setting of the B or X location counter.

When MAC is loaded, the X and B location counters are set to "zero" (effectively). Thus, programs written using the old method of accessing locations relative to the B and X register need not be changed. The)CLEAR command also "zeros" the X and B location counters. To explicitly zero the X or B location counter at assembly time, give the command

```
)9SET,B  ZRO
```

or

```
)9SET,X  ZRO
```

To examine the *effective* contents of the B location counter, give the command

```
400-,B  :
```

To examine the *effective* contents of the X location counter, give the command

```
1000-,X  :
```

)WLOC

The)WLOC command prints out all of the user defined symbols on the device associated with the list stream. Six symbols are printed on each line.

)WMNE

The)WMNE command prints out the operation codes and addressing mode specifiers that are built into MAC on the device associated ith the list stream. Six symbols are printed on each line.

)WRITE

The)WRITE command prints a list of symbols and their values on the device associated with the list stream. The symbols to be printed are specified following the)WRITE command and are delimited by spaces. The list of symbols is terminated by the end of the line. Only user defined symbols will be printed and symbols used but not yet defined are printed with the characters "NF" (not found) instead of a value. Four symbols are printed on each line.

)WRTM

The)WRTM puts MAC in write mode, that is, the)WRITE command is not ignored. The)NWRT command puts MAC in non-write mode. MAC is initially in non-write mode.

)WRUS

The)WRUS (write undefined symbols) command causes all symbols used but not yet given values to be printed, in order of their first appearance, to the list stream. This command may be abbreviated with a question mark (?).

)9ASSM

This command is used to manipulate file names or numbers associated with the source, list and object streams. The command may be followed up by up to 3 arguments, each representing a file name or number. The arguments, if any, must be separated by commas and terminated by carriage return. For example,

```
)9ASSM   INFIL,0,UTFIL
```

If a file name is omitted, then the previous file defined by an earlier)9ASSM or $ command is used. The OPEN, CLOSE and CREATE procedures are automatically taken care of by MAC. When a file associated with a specific stream is replaced, MAC opens the new file and closes the old one. If a file name is included in double quotes the file is created before it is opened. Some examples follow.

```
)9ASSM   (SANNER) SYMBOLFIL
)9ASSM   ,,"BPUNFIL:BPUN"
)9ASSM   ,0
)9ASSM
)9ASSM   SYMPROG, L-P, "BRFPROG"
```

The table below contains the additional information which is supplied by MAC when opening files.

File Connected to:	Default Type:	Access Code:
Source stream	:SYMB	RX
List stream	:SYMB	W
Object stream	:BRF	W

)9EXIT

This command returns control to the operating system and all files are closed. Note that the stream specifications are unchanged if you restart MAC with the @CONTINUE command.

)9PARI

This command acts as a switch and turns off/on parity checking on input. The parity checking is initially on, i.e., an error message is given if an ASCII character is encountered in the source input stream having odd parity.

)9ASCI

This command causes the contents of memory to be dumped on the file connected to the list stream. The format is ASCII, i.e., each word is converted to two ASCII characters. The dump area corresponds to the interval last specified by the < command.

)9LITR

)9LITR acts as a switch and turns on/off the duplication of literals in the literal dump area ()FILL). Initially, literals are *not* duplicated in order to save memory space, i.e., each use of the same literal allocates the same memory location. However, duplication of literals may appear to advantage when debugging programs. There are some limitations in this optimization feature:

1. no optimization in two-pass assembly
2. no optimization upon literals with multiple location values

3.2.3.10 Conditional Assembly (")

The "command sets MAC in conditional assembly mode. MAC then expects a logical expression terminated by carriage return. The " immediately followed by carriage return means reset conditional assembly mode.

The expression must consist of symbols separated by logical operators. The value of the symbol is true if the symbol is located in the undefined symbol table (referred, but not defined). Else the value is false. Such symbols are often called library marks.

The logical operators used with the " command are:

(space) AND (∧)
\+ (plus) OR (V)
— (minus) NOT (⌐)

Expressions are evaluated from left to right and use of parantheses is not allowed. An expression may begin with the negation operator. If the value of the expression is true, the text up to the next " is assembled. If the value is false, the text up to the next " is ignored.

Example:

```
"—PER                   % NOT PER
———————
———————                 % TEXT
———————
"OLE+PER                % OLE OR PER
———————
———————                 % TEXT
———————
"
```

4 EXTENDED MAC

4.1 *OPTIONS*

MAC is orginally a number of different systems, each having its own configuration of facilities. This section describes all of the possible additional facilities, known as options.

The option concept has survived from the time of expensive memory, but is not really legitimate today! The MAC assemblers supplied with NORD-10/SINTRAN III all include the "options" which are described in the following sections. Symbols related to options are made permanent in the local symbol table.

4.2 *THE OPTIONS AND THEIR USE*

4.2.1 *Binary Relocatable Format Output (BRF)*

An assembler providing no other option than absolute assembly would be quite constraining for the user. It would be difficult to link programs written in assembly language to programs written in other languages and vice versa, each time a library subroutine was to be used it would have to be completely reassembled, perhaps a time consuming process, etc. Consequently, it has assembly output which removes the above contraints. This "option" is *not* really an option at all. It is implemented as a part of basic MAC. However, we delayed its description until this section to simplify the description of basic MAC.

4.2.1.1 Summary of Usage

The alternate form of assembly output that was decided upon is a traditional one: binary relocatable format output (BRF) which is loadable by a suitable linking relocating loader. For readers unfamiliar with the terms binary relocatable format and linking relocating loader, we will briefly explain these terms and the concepts behind them.

There are fundamentally two concepts to explain: the concept of relocatable output as contrasted with absolute output and the concept of linking. The reader is undoubtedly already familiar with absolute output as this is what MAC normally produces. With absolute output, each instruction or constant is completely assembled so that it is the correct content of a particular, predefined memory location. In other words, the instruction or constant is ready to execute once it is loaded into the correct memory location. This instruction or constant is assembled directly to memory in absolute mode assembly, and programs may later be dumped in absolute binary format ()BPUN) or octal format ()PUNCH).

Relocatable output on the other hand is output produced in a format suitable for later being loaded anywhere in memory, and is connected to the object stream when the)9BEG command is executed. The essential feature of this output format is that addresses may be relative to the beginning of the program rather than relative to a particular memory location. Thus, the absolute code

```
        LDA     FOO
        JMP     *
FOO,    *–2
```

would be assembled to

```
0/      LDA     FOO
        JMP     *
FOO,    *–2
```

and would be output including indications that when the code was later loaded, starting at location 1000 say, it should be modified so it is just as if

```
1000/   LDA     FOO
        JMP     *
FOO,    *–2
```

had originally been written. The reader will immediately see that a complete relocatable output and loader system would permit the following:

1. absolute contents of an absolute location
2. absolute contents of a relocatable location
3. relocatable contents of an absolute location
4. relocatable contents of a relocatable location

MAC is capable of producing the desired results in all four cases.

In order for MAC to "mark" its output so it will correctly be either relocated or not relocated at load time, when MAC is in BRF mode each symbol is defined to be either *fixed absolute* or *relocatable*. Unless otherwise declared, all symbols declared via the comma command, e.g.,

FOO,

are taken to be relocatable. But even symbols thus declared can be explicitly declared to be fixed absolute ()9FABS).

Symbols declared with the equal command, e.g.,

A = 3

are declared to be either fixed absolute or relocatable depending on whether the expression on the right side of the equal sign is fixed absolute or relocatable. All constants are fixed absolute. If B is declared to be equal to FOO by the command

B = FOO

where FOO was defined as above, B is relocatable. Making

A = B – C

where both B and C are relocatable, makes A absolute. If only B was relocatable, A would also be relocatable.

The concept of linking allows programs assembled or compiled at separate times to be loaded into memory and *linked* together. Thus, it is possible to write a program which references symbols in other programs which are assembled separately and to load the programs in such a way that they actually work.

A number of commands have been added to MAC to facilitate production of relocatable, likable code. These are discussed in detail later in this section, but four commands which are particularly important will be discussed immediately.

)9BEG puts MAC in a mode where it produces relocatable, linkable output which it writes to the object stream.

)9END resets MAC so it produces absolute output which it writes to memory.

)9EXT declares symbols to be external to the particular program being assembled. In other words, symbols declared with the)9EXT statement will be found in another program and a suitable linkage must be made at load time.

)9ENT declares symbols which may be referenced as external symbols from other programs. Thus,)9ENT is the complement of)9EXT.

Programs starting with)9BEG and ending with)9END are called program units or BRF units and may be subprograms as well as main programs. Program units and communication between them using)9ENT and)9EXT is illustrated below.

```
)9BEG     MAIN
)9EXT     SUBR  DATA
MAIN,     —
          —
          —
JPL  I    (SUBR
          —
          —
          —
LDA  I    (DATA
          —
          —
          —
)9END
```

Main program unit

```
)9BEG
)9ENT     SUBR
SUBR,     —
          —
          —
)9END
```

Subprogram unit

```
)9BEG
)9ENT     DATA
          —
          —
          —
DATA,     3
          —
          —
)9END
```

Subprogram unit

Program units are later loaded anywhere in available memory and linked together with a loader. Available subsystems which are able to perform the loading and linking procedure:

- SINTRAN III Real-Time Loader
- NORD-10 Relocating Loader

These are described in the respective manuals.

Before discussing the commands related to BRF assembly we recommend the reader to read the following lines carefully:

- Undefined symbols are always taken to be relocatable. Thus, assembly in some cases will be different from the programmer's intention, as in this example:

```
A, 0          % Relocatable definition
A-DELTA       % Absolute! Meaning: relocatable
A + DELTA     % Illegal expression! Meaning: relocatable
DELTA = 20    % Absolute definition
```

- The warning "possible fault" is *not* given in BRF output mode, even in situations where the message appears in absolute assembly mode.

  Example:

```
100/      LDA FOO
* + 200/      FOO, 0        % Range is exceeded, but no message!
```

These problems are avoided by using the two-pass assembly option.

4.2.1.2 Commands Included with the BRF Option

In this section we discuss in detail each of the commands included with the BRF option. Some of them are implemented in order to communicate with high level program units written in FORTRAN or BASIC. Therefore, it is supposed that the reader has some knowledge about real-time programming and layout of the COMMON area.

)9BEG

The)9BEG command has already been discussed briefly. The command instructs MAC to switch to BRF output mode.)9BEG must be followed by carriage return, or a space and one label, as in

)9BEG FOO

This label is called a main entry other than entries declared by)9ENT and must be defined by the user at the instruction he wants his program to start (by the RUN command in the loader). Main entry definition is somewhat different for real-time programs (see)9RT).

)9BEG also causes MAC's location counter to be set to one (one may seem a strange value - "why not zero?", the astute reader will ask. However, it is really set to one but the loader corrects this "mistake").

)9ENT

As already stated, this command declares the following symbols to be entries. For example,

)9ENT PER1 FOO ZOO

Entries must later be defined by comma (,) or '' = *''.

NOTE: The symbols are placed in the local symbol table at the time of the)9ENT command. Address references are updated when the symbols are defined (,).)PCL should therefore not be used in connection with "entry" symbols.

)9EXT

The symbols declared with the)9EXT command are declared to be external symbols. For example,

)9EXT LUCY SNOOPY CHARLIE BROWN

Externals must later be referenced indirectly (not by displacement!). Due to limitations in the loader, address arithmetic is *not* permitted upon externals.

)9END

This command puts MAC back to absolute assembly mode. All symbols defined since the last)9BEG are deleted from the symbol table. Symbols referenced since the last)9BEG but undefined are printed on the list file, and at last a checksum is generated which is checked by the loader. If MAC was already in absolute assembly mode when the)9END command is given, symbols defined since the last use of the)9ENT command are deleted from the symbol table.

)9FABS

The)9FABS comand takes a list of previously defined symbols as arguments. For instance,

)9FABS AUD BJØRG BRITT MARIT UNNI

Each of these symbols are declared to be a fixed absolute rather than a relocatable symbol and will not be relocated at load time.

)9EOF

This command will cause an end-of-file mark to be written to the object file thus terminating a sequence of BRF program units.

)9LIB

This command takes a list of library entry symbols as arguments. For example,

)9LIB SQRT

The command is used *within* library program units (delimited by the)9BEG and)9END) as follows:

```
)9BEG
)9LIB     SIN COS
)9ENT     SIN COS
SIN,      ...
          ...
COS,      ...
          ...
)9END
```

When the object code representation of)9LIB is detected within a program unit by the loader, the code up to the next)9END is loaded if at least one of the symbols associated with the)9LIB command is undefined in the loaders symbol map. If not, the code up to the next)9END is ignored.

)9ASF

COMMON blocks are normally defined by a high level language program, but if desired also in assembly (real-time applications).

This command takes two symbols as arguments, a COMMON label and a symbol which defines the length of the common area. The latter symbol must be fixed absolute. For example,

```
SIZE = 144
)9ASF   BLOK1   SIZE
```

The object code of this command will cause the loader to reserve 144_8 locations in the common area. This COMMON block starts at label BLOK1.

)9ADS

The command described above defines a COMMON block, however,)9ADS is used to access variable(s) within a given COMMON block from assembly.)9ADS takes two symbols as arguments, a COMMON label and a symbol representing a displacement relative to the COMMON label. A blank COMMON is accessed by using the symbol BLANK. Note that the location counter is incremented by one as MAC assembles the displacement into this location. At load time the address of the COMMON label is added to this displacement. The command may be preceded by a MAC label definition as in the example:

High level language definition:
COMMON/BLOK2/IAR(10)

Assembly access:

```
DISP = 0
BLOKAD, )9ADS   BLOK2   DISP
START,    LDX   BLOKAD        % Address of BLOK2
          LDA   ,X            % First element, IAR(1) if FORTRAN
          —                   % First element, IAR(0) if BASIC
          —
```

)9LC

This command is also used to form a COMMON address and takes one symbol as an argument. For example,

```
DISP=0
LOWAD,  )9LC DISP
START,  LDA LOWAD
        AAA — 1          % upper bound
```

The location counter is incremented by one as the value of the symbol is assembled into this location. The command is much like)9ADS, but at load time the *lowest* COMMON address is added to this displacement.

)9RT

This command declares a program unit to be a real-time program with a desired priority. It takes two symbols as arguments; the first is the name of the real time program (main entry), and the second specifies the priority. For example,

```
PRIO=35
)9RT SYSA PRIO
```

The command must precede the main entry point of the program, thus making redundant the declaration described under)9BEG. When the object representation of)9RT is detected by the RT loader a so-called RT description is generated. Further information is found in the SINTRAN III User's Guide.

4.2.2 *Standard Tables*

This option is presented independent of basic MAC, but is really a necessary part of a complete assembly system. The size of the three main tables in MAC may be changed by the)9TABL command (Section 4.2.9.2). Standard sizes of these tables are:

7000_8 entries in local symbol table*
200_8 entries in constant table (literals)
4000_8 entries in undefined symbol table

* Does not include the fixed symbols. The number of fixed symbols depends on the number of options added.

4.2.3 *The ZERO, CORE, LIST, PCL and CHANGE Commands*

This option adds five commands to MAC; the)ZERO,)CORE,)LIST,)PCL and)CHANGE commands. We will discuss each of these commands in turn.

4.2.3.1)ZERO

First note that this command is a dummy command in BRF mode assembly. The)ZERO command sets to zero all memory locations in the interval specified by the < command. For example, the commands

```
300 < 400
)ZERO
```

will set all memory locations in the inclusive interval 300 through 400 to zero. If the command is terminated by carriage return, the above described action is taken. However,)ZERO may be followed by a space and one symbol (mnemonic or previously defined) which value is added to the zero.

4.2.3.2)CORE

The)CORE command prints to the list stream the upper and lower bounds of the areas of memory used by MAC. For example,

```
)CORE
SOFTWARE PROTECTED AREA    063377-072407
TABLE AREA 033775-056711
```

indicates memory locations 63377 through 72407 are occupied by basic MAC and locations 33775 through 56711 are occupied by the symbol tables.

4.2.3.3)LIST

The)LIST command ouputs the symbols in the local symbol table (i.e., the symbols not built into MAC) to the object stream device. If this device is the terminal, the symbols and their values will be printed in the format:

```
)LIST
    MAC=037777
     PER=000050
       A=000501
     FOO=002713
@
```

If the object stream is connected to a file, the symbols will be able to be reloaded at a later time. Why the object stream? The absolute binary memory dump produced by)BPUN is logically connected to this stream, and)LIST is often output to the same file. Thus, later loading will restore the user to the point where the memory and symbol dump was taken.

4.2.3.4)PCL

The)PCL, or partial clear, command expunges from the local symbol table all symbols which were defined later (in time) than the symbol specified with the)PCL command. Symbols thus expunged may be reused: it is as if they had never existed. For instance:

```
400/
S1,         STA       SAVE
S2=403
            LDA       B
S3,         COPY      SL  DA
            JMP *
SAVE,       0
B,          0
)PCL S2
```

After execution of the)PCL command above, the symbols S1 and S2 will be defined and the symbols S3, SAVE and B will not exist.

4.2.3.5)CHANGE

The)CHANGE command causes MAC to search an area of memory for all memory locations which match a certain constant in some specified bits. In each memory location in which a match is found, the specified bits are changed to a new constant. This command is used as follows: make the contents of the memory location labeled OLD be the constant that is sought. Make the contents of the memory location labeled NEW to the new constant. Make the contents of the memory location labeled MASK be an octal mask in which bits that are one indicate bits to be checked for a match and changed. Set up an interval using the < command. Execute the)CHANGE command.

For Example:

```
OLD/        000177
NEW/        000000
MASK/       000777
2000 < 3000
)CHANGE
```

will change to zero the low nine bits of all memory locations in the inclusive interval 2000 through 3000 which are found to contain 177 in the low nine bits. The memory locations labeled OLD, NEW and MASK are added to MAC as part of this option.

4.2.4 *Breakpoint*

The breakpoint option allows an executing program to stop immediately before the instruction in any preselected memory location is executed so the contents of the computer's registers can be looked at. This is a very useful debugging aid. Generally three commands are used in connection with breakpoints. These commands are the new period (.) command, the / command and the ! command. Their use is illustrated below. Suppose the program,

```
400/
            LDX     M6
            LDA     ZEERO
            AAA     1
            AAA     1
            JNC     *–2
            JMP     *
M6,         177722
ZEERO,      0
```

has been assembled and the command

403.

has been given. This latter command tells MAC to stop program execution just before the second AAA instruction has been executed. If the program is now started with the command

400!

the LDX, LDA, and first AAA instructions will be executed and MAC will print a period preceded and followed by carriage return and line feed, and await further commands. Any command may now be given; however, to look at the contents of one of the registers, give the command

Bn/

where n selects a specific register. Thus, using A in the above example will print the value 1, the current contents of the A register.

BA/000001

and BX will print 177772

BX/177722

that is, –56.

Suppose the breakpoint is now moved to the JMP instruction with the command

405.

The command

M6–1.

would have had the same effect, and in fact, the argument preceding the . can be any completely defined expression. Execution may now be continued by giving the command

!

Alone, ! causes execution to commence with the instuction upon which the preceding breakpoint was set. Thus, after the three-instruction loop above has been executed 55_8 more times, MAC will again print a period preceded and followed by carriage return/ line feed, and await further commands. Now the command BA will print a 134

BA/000134

and BX will print a zero

BX/000000

The other registers could be examined with the

BD/
BT/
BL/
BP/
BSTS/
BIR/

commands. The contents of the A, D, T, X, B and L registers can be changed after the contents of the register has been examined, just as normal memory locations can be changed using the / command. For example,

BA/000006 7

will change the contents of the A register from six to seven.
1!

has a special meaning. Suppose in the example above, after examining the contents of the A and X registers after the instruction in memory location 403 has been executed the first time, the

1!

command is given. This will cause one instruction to be executed and MAC to await commands again. Now the

BA/000002

command will print two. Thus

1!

provides a way of stepping through successive instructions in a program. Continuing with the same example, the command

1!

will move the breakpoint to the JMP * instruction and cause the JNC *–2 instruction to be executed which will cause a jump back to the first AAA instruction, and a number of instructions will be executed before the breakpoint on the JMP * instruction is reached. The command

0.

also has a special meaning when breakpoints are active. It removes the breakpoint if it is set, and execution will continue without breaks when the next ! is given.

Several breakpoints (maximum 10) may be specified if MAC runs under control of the SINTRAN operating system.

Since 0. and 1! are special commands, it is illegal to specify breakpoints in locations 0, 1 and 2 of memory. In stand-alone systems these locations are even used to execute a break.

4.2.5 *Decimal Mode*

This option gives MAC the capability of treating integer numbers which it reads and writes as decimal as well as octal. This option adds two commands to MAC,)DEC and)OCT. These two commands set MAC in either decimal or octal mode, respectively. When MAC is in octal mode, as it is initially, integers are treated as octal as described in the chapter on basic MAC. When MAC is in decimal mode, integer numbers are assumed to be in the conventional decimal format, that is, signed numbers consisting of the digits zero through nine. When in decimal mode all numbers are printed as decimal with leading zeros elided and zero printed as 0. Thus, if MAC read the following input

```
)OCT
400/
            10
            177770
)DEC
            8
            -8
```

locations 400 through 403 would contain the octal contents

```
)OCT
400/000010
401/177770
402/000010
403/177770
```

Now the following commands will have the effects shown

```
)OCT
400<403
)PRINT
000400/000010
177770
000010
177770
@
)DEC
)PRINT
256/8
 -8
8
 -8
@
```

4.2.6 *Floating Point Numbers*

This option adds to MAC a facility for accepting and printing floating point numbers. See section 2.1.4 for 48 bits floating point numbers (standard) and Appendix E for 32 bits floating point numbers (optional). Readers being concerned with the latter must note that this format occupies two memory locations when reading this section.

The output format from MAC for floating point numbers is always the same. We can best describe it with some examples:

```
 3.12000000E−01
−2.61234460E+03
              0
 2.60000000E+50
−2.40000000E−50
```

This is a form of the conventional "scientific" number notation. That is, the above is a way o writing

$3.12 * 10^{-1}$
$-2.6123446 * 10^{3}$
0
$2.6 * 10^{50}$
$-2.4 * 10^{-50}$

Zero is the only number printed in an exceptional way. For those readers familiar with FORTRAN, floating point numbers with the (possible) exception of zero are printed according to an E15.8 specification.

The MAC routine which converts the internal binary format of floating point numbers to the external decimal format rounds to the nearest correct decimal digit.

The input format for floating point numbers is "free". Some examples of floating point numbers acceptable to MAC are:

```
 1.1
 1
−1
 0.000001234E15
321.1234
−E1
 1.
 E
```

There are two ways of getting MAC to accept a floating point number: using the [command and using the \ command. We discuss the [command first. The [command instructs MAC that a floating point number follows which MAC is to convert into internal format and send to the object stream as three consecutive words. The location counter is then incremented by three. Some examples of the use of the [command are:

```
)FILL
400/
            LDF     ([ 3.2
            FAD     FOO
            JMP     *
FOO,        [ 4.5
)FILL
```

This will result in the floating point constant 4.5 being placed in locations $403\text{-}405_8$ and the floating point constant 3.2 being placed in locations $406\text{-}410_8$. When floating point numbers are written in literals on the same line as a floating point instruction, it is not necessary to include the [. Thus,

```
          LDX     (–6
          LDF     (7
          FMU     (6
          FDV     (3.2E2
          JMP     *
)FILL
```

is identical to writing,

```
          LDX     (–6
          LDF     ([7
          FMU     ([ 6
          FDV     ([ 3.2E2
          JMP     *
)FILL
```

The \ command should generally be used only on-line. This command is similar to the / command but prints the contents of a specified memory location (three memory locations) as a floating point number instead of as an octal integer. For example:

```
PER \ 3.14159260E+00
PER+3\              0
PER+6\–1.00000000E+11
```

MAC's output is underlined. Once a floating point number has been printed using the \ command, the number may be changed by typing a floating point number preceded by [and followed by a carriage return. Thus,

```
PER\3.14159259E+00      [10.1
PER\1.01000000E+01
```

will change the content of location PER, PER+1 and PER+2 to 10.1. If a floating point number is printed using \ , but it is not desirable to change it, only a carriage return should be typed.

```
PER\1.01000000E+01
PER\1.01000000E+01
```

If a change is made, the location counter is advanced by three. If no change is made, the location counter remains unchanged.

```
400\1.01000000E+01      [01
*:000403

400\1.00000000E+00
*:000400
```

4.2.7 *Disassembler*

This option provides a way to print the content of a memory location out in a symbolic format. This is very useful when examining and changing instructions while debugging a program. This option adds two new commands to MAC:)SETSM, which switches MAC into symbolic printout mode, and)RESSM, which switches MAC back to octal printout mode. An example will clarify the use of the disassembler option. Suppose the code,

```
400/
            LDA     FOO
            STA     ZOO
            JMP     *–2
FOO,        0
ZOO,        0
```

has been assembled. Now we switch to symbolic output mode using

```
)SETSM
```

Now we examine the content of locations 400 through 404, like this:

```
400/ LDA *003
401/ STA *003
402/ JMP *–002
403/ STZ *000
404/ STZ *000
```

or like this:

```
400< ZOO
)PRINT
400/LDA *003
STA *003
JMP *–002
STZ *000
STZ *000
@
```

The disassembler's symbolic output has the following properties:

1. The displacement is written as a three digit octal number.
2. All numbers are written as signed three digit integers.
3. In a ROP or SKP instruction the disassembler will write a zero if the source or destination is not specified.
4. MIS instructions are not translated. These and other unacceptable numbers will be written as six digit *octal* numbers.
5. Currently, NORD-10 mnemonics (not in NORD-1) are *not* translated.

4.2.8 *Two-Pass Assembly*

This option gives MAC the capability to execute assembly in two passes, i.e., the source code is scanned twice. Thus, it is possible to use MAC in three modes: "debug mode", "1-pass mode" and "2-pass mode". Assembly in two passes will eliminate the warning "POSSIBLE FAULT" as all symbols are defined in pass 1. The assembly listing generated in pass 2 is extremely suitable for later debugging: an example is given in Figure 4.1, following. Listing may, however, grow very large, accordingly experienced users may alternatively use the)9SCLC or)9SLPL options which are described later.

Assembly in two passes will normally follow this procedure: set actual value to the current location counter and type the command

)1PASS

on-line. Then assemble the source program (pass 1). Next, type the command

)2PASS

on-line, and assemble the source program again (pass 2). Two-pass listing is connected to the list stream. MAC may be reset to "debug mode" by the command

)9DBUG

Example:

```
301/000000
)1PASS
)9ASSM FILAMI
**** 000000 DIAGNOSTICS ****
)2PASS
)9ASSM FILAMI, L-P, OBJECT
**** 000000 DIAGNOSTICS ****
)9DBUG
```

Operating in "debug mode", MAC will work as usual (undefined symbols are allowed).

Output generated by MAC in pass 1 is inhibited; object output as well as output from)WRITE,)LIST, etc. However, error messages are written. One should note that programs with extensive use of)KILL and/or)PCL require great capacity of the local symbol table, because all symbols are stored in pass 1.

The final assembly takes place in the second pass. This pass expects that all symbols are known. Unrecognized symbols will, however, be stored in the undefined symbol table. If messages are given in pass 1, check if there were serious error messages or warnings. At)2PASS the current location counter is automatically reset to the same value as with the)1PASS command. Error messages in pass 2 are written on the line preceding the erroneous line.

We have experienced that the following points should be emphasized, so please note:

— Only one BRF unit should be involved in a two-pass assembly.

- No symbols are stored in the undefined symbol table during pass 1. Therefore, symbols used with *conditional assembly* ('') must exist in that table before typing the)1PASS command.

- The expression PER = OLE demands the OLE is defined at '' = '' in pass 1.

- The message ''ALREADY DEFINED'' is not issued during pass 1. It is the user's responsibility that symbols do not conflict with built-in symbols of the local symbol table (refer to Appendix B.2).

The seven fields of the two-pass listing contain this information:

Field 1: Current location counter.

Field 2: Instruction code for memory reference and argument instructions.

Field 3: Addressing modification for memory reference instructions given as a combination of the letters X, I, B and space.

Field 4: A) Effective location for P-relative memory reference instructions, given as a 5 digit octal number.

B) For not P-relative memory reference instructions, the displacement is given as a 3 digit signed octal number.

Field 5: Final machine code in octal form.

Field 6: Space or one of the following letters:

R means relocatable expression, given only in BRF assembly.

E means external, given only in BRF assembly.

U Unrecognized symbol. It is stored in the undefined symbol table with correct address reference.

W Warning — command is not executed.

Field 7: Original symbolic assembly code.

```
                                        % DECISION ROUTINE NO 1
                                        )9BEG
                                        )9ENT MAIN
                                        )9EXT SUBR1 SUBR2 NEXT PPP1
                                        DIFF=-105
                                        SYSTEM=301
                                        TEST=0

00001                    000000   M3,    0
00002                    000000   M2,    0
00003                    000000   M1,    0
00004                    000000   MP,    0
00005                    000000   P1,    0
00006                    000000   P2,    0
00007                    000000   P3,    0

                                        )9SET ,B MP

00010 044       00036 044026   MAIN,  LDA    (MP
00011                  144053          SWAP   SA DB
00012 014    B  -003 014775          STX    M3 ,B
00013 030    B   001 030401          STF    P1 ,B
00014                  150002          TRA    OPR
00015 170              171400          SAX    0
00016 130       00020 130402          JAN    * 2
00017 134  I    00037 135020          JPL I  (SUBR1
00020 050       00035 050015          LDT    VAR1
00021                  146067          RADD   ST DX
00022 130       00027 133005          JXZ    CONTI
00023 054 X      -105 056273          LDX    DIFF ,X
00024 044  IB    -003 045775          LDA I  M3 ,B
00025                  146675          RSUB   SX DA
00026 070       00040 070012          AND    (77777
00027 134  I    00041 135012   CONTI, JPL I  (SUBR2
00030 054  I    00042 055012          LDX I  (PPP1
00031 034    B   001 034401          LDF    P1 ,B
00032                  175057          BSKP   ZRO TEST DX
00033 124  I    00043 125010          JMP I  (SYSTEM
00034 124  I    00044 125010          JMP I  (NEXT

00035                  015367   VAR1,  15367
00036                  000004 R )FILL
00037                  000000 E
00040                  077777
00041                  000000 E
00042                  000000 E
00043                  000301
00044                  000000 E

                                        )9END
```

Figure 4.1: Example of a Two-Pass Listing

4.2.9 *Macros*

This option gives MAC a simple macro capability.

4.2.9.1 Introduction to Macros

For readers unfamiliar with the concept, the most basic use of macros is for abbreviating a multi-line sequence of code or data with a single line. For instance, it is quite probable that within a program the code sequence to load the A register from memory and then store the A register to memory might often be used. For example,

```
LDA     A
STA     B
```

These two instructions logically perform a memory-to-memory "move" instruction. For increased readability or to save a few characters in the source program, it might be desirable to abbreviate these two lines to the single line

```
MOVEAB
```

Macros permit such an abbreviation. If, somewhere before the first use of the MOVEAB abbreviation, the following lines of code are written,

```
)MCDEF   MOVEAB
         LDA     A
         STA     B
]
```

the MOVEAB abbreviation becomes valid. The above lines of code are called the macro definition for the macro MOVEAB. After the macro MOVEAB has been defined, the line of code

```
MOVEAB
```

is called the macro call. Each time the macro MOVEAB is called, it is just as if the lines

```
LDA     A
STA     B
```

has been written instead. Thus, if MOVEAB is defined as above, then writing

```
MIN     A
MOVEAB
JMP     *
```

is exactly the same as writing

```
MIN     A
LDA     A
STA     B
JMP     *
```

Similarly, a macro could be used to abbreviate a data sequence. For instance, if

```
)MCDEF    DATA1
          123456
          0
          123456
]
```

is first written, writing

```
FOO,      1
          DATA1
          2
```

later is exactly the same as writing

```
FOO,      1
          123456
          0
          123456
          2
```

Let us now return to the example of the macro to abbreviate a move sequence. Suppose moves between different locations are desired, e.g.,

```
          LDA       A
          STA       B

          LDA       D
          STA       E

          LDA       PER
          STA       OLE
```

Would it not be convenient to be able to specify at the time of the call of the MOVE macro between which locations the move should be done? For instance,

```
          MOVE      A,B
```

becomes

```
          LDA       A
          STA       B
```

While

```
MOVE      PER,OLE
```

becomes

```
          LDA       PER
          STA       OLE
```

Macros permit such parameterization of the macro call. A proper macro definition of the macro MOVE to perform the above function would be:

```
)MCDEF   MOVE    $X, $Y
         LDA     $X % always an extra space after dummy parameters
         STA     $Y % extra space
]
```

In the above definition parameters $X and $Y are dummy parameters, place holders, which indicate where arguments of the macro's call should be substituted. Thus,

```
         MOVE    ABCD,EF
```

is the same as writing

```
         LDA     ABCD
         STA     EF
```

Another useful macro to define might be the MDC (memory decrement) macro,

```
)MCDEF   MDC     $L
         LDA     $L % extra space
         AAA     -1
         STA     $L % extra space
]
```

Subsequently writing

```
         MDC     PER
         MIN     PER
```

would leave the value of PER unchanged.

4.2.9.2 Defining and Calling a Macro

A rigorous specification for the definition and use of macros follows:

- A macro must be defined before it can be called.
- The macro definition consists of three sections: the macro definition line, the macro body and the macro termination character.

```
Macro definition line ──────► )MCDEF SUM $A, $B, $C
                              LDA $A % extra space
Macro body ─────────────────► ADD $B % extra space
                              STA $C % extra space
Macro termination character ► ]
```

- The macro definition line consists of the command)MCDEF followed by at least one space, followed by the macro name and an optional dummy parameter list. The macro name may be any previously unused MAC symbol. the dummy parameters must be MAC symbols preceded by the character $. The MAC symbols used in a dummy parameter may have been previously defined. Use in a macro dummy parameter in no way affects a previous definition of a symbol. The elements of the dummy parameter list are separated by commas, and a maximum of 20 dummy parameters may be specified in a macro definition. The first carriage return after the)MCDEF terminates the macro definition line.

- The macro body starts immediately after the macro definition line and extends to the termination character.

- Dummy parameters within the macro body must be followed by a space or carriage return to be recognized. This following character will not be included in the macro body when the macro is called, i.e., for correct assembling, a statement ending with a dummy parameter must be terminated by space and carriage return.

- Macro calls always consist of exactly one line containing the macro name and an optional actual parameter list. For example,

```
SUM     A,B,C
```

  The macro name must be followed by a space, and actual parameters are separated by a comma. The macro call is terminated by the first carriage return after the macro name. Actual parameters may consist of any characters with the exception of comma and carriage return. For example,

```
SUM     60—A, (340, I  FOO
```

  is the same as writing

```
LDA     60—A
ADD     (340
STA     I  FOO
```

4.2.9.3 Related Commands

The macros option includes another command besides)MCDEF and changes the meaning of the : command. The new command is)LSTM which is connected to the list stream. It lists the names of all defined macros, one name per line. The : command when preceded by a macro name, lists the macro definition, but the dummy parameters will be replaced by the symbols:

$1, $2, $3, $4, $5, $6, $7, $8, $9,
$:, $;, $<, $=, $>, $?, $@, $A, $B

For instance,

```
SUM:    LDA $1
ADD     $2
STA     $3
]
```

In order to get the macro definition listed in a tabulated format, each dummy parameter in the macro body may have to be followed by a space.

The)KILL command will delete a macro name and its associated definition. The)PCL command should not be used when there is a macro name within its scope.

4.2.10 *The 9READ, 9TABL, FIX, 9SCLC, 9RCLC, 9SLPL, 9RLPL Commands*

This option adds seven commands to MAC: the)9READ,)9TABL,)FIX,)9SCLC,)9RCLC,)9SLPL and)9RLPL commands.

4.2.10.1)9READ

The)9READ command reads binary information produced by the)BPUN command. The information is placed in memory according to the limits specified for)BPUN. The octal part is ignored, i.e., the exclamation mark is searched. Note that the command takes the *input* from the file last connected to the object stream unless an optional parameter is supplied. The parameter must be the file name or number of the binary input file.

Examples:

```
1,0,101$                % must be opened for "RX" outside MAC
)9READ
1,0,3$                  % reset object output (101 is closed)

)9READ   BINFIL1        % opened for "RX" by MAC, not closed
)9READ   104            % must be opened for "RX" outside MAC, not closed
```

4.2.10.2)9TABL

)9TABL enables the user to change the size of the three main tables in MAC. The current size of the tables may be examined by the)CORE command. One should be aware that each table entry (one symbol) occupies three memory locations.)9TABL must be followed by three symbols separated by space, and terminated with carriage return. The value of each symbol is taken to be the *number of entries* for the tables in this order:

1. Local symbol table (user defined symbols)
2. Constants table (literals)
3. Undefined symbol table

Fixed symbols in the local symbol table are not included. The)9TABL command also clears the tables.

Example:

```
)CLEAR
A=1000          % number of entries in local symbol table
B=62            % number of entries in constants table
C=600           % number of entries in undefined table
)9TABL  A B C
```

4.2.10.3)FIX

The)FIX command makes all symbols in the local symbol table permanent. That is, they will not be deleted by any)CLEAR command given later on.)FIX is intended to be used to make global (or system) variables permanent when assembling a system from parts.

4.2.10.4)9SCLC and)9RCLC

)9SCLC switches MAC to a mode which prints the value of the current location counter on the device associated with the list stream. The value appears as a six digit octal number and two spaces preceding the original symbolic code. This mode is ignored when MAC is operated on-line. The mode is reset by the)9RCLC command.

4.2.10.5)9SLPL and)9RLPL

)9SLPL is implemented to be used when assembling NORD-PL object code. The printout on the list stream will contain the value of the current location counter followed by the NORD-PL source statement(s). The NORD-PL program must be compiled like this:

@DEV INPUT, OBJECT, OBJECT

thus having a % preceding the source line. In this mode MAC inhibits printing of anything but the text between % and line feed.

)9RLPL turns off the mode described above. Note that)9SLPL simulates)9RCLC and)9SCLC simulates)9RLPL.

4.2.11 *The TRACE*

The TRACE is too comprehensive to be described in this manual, therefore those interested should consult the manual "The TRACE Routine" (ND-60.046).

The purpose of TRACE is to be a tool when debugging programs written in any language, but of course compiled to fit the NORD-10 instruction set.

TRACE executes the other program instruction by instruction, thus having the control all the time.

Various protection conditions may easily be set through MAC commands and when violated control is passed to MAC. An extensive log is produced (if desired) which may be connected to any file.

The commands of TRACE are listed below:

)9INN	Set start point of tracing
)9STOP	Set stop point(s)
)9TPO	Set trace point(s)
)9PPO	Set printing point(s) (where memory area(s) are dumped)
)9LAR	Set legal area(s)
)9TAR	Set tracing area(s)
)9BAR	Set "blocked" area(s)
)9MAR	Set memory dump area(s)
)9ROT	Set rotation counter and address
)9DS	Set disassembly of current instruction
)9RDS	Reset disassembly of current instruction
)9STEP	Set step counter
)9IOT	Set IOT simulation
)9FLT	Print floating accumulator as a floating point number
)9TR	Start tracing
)9CON	Continue after a stop
)9REG	Set register print switches
)9NVAL	Examine/change new register values
)9OVAL	Examine/change old register values
)9PRIV	Set action to take if a privileged instruction is encountered.

5 USING MAC

As emphasized in the introduction of this manual, the concept of MAC allows the user to be extremely free in his composition of the assembly source programs as well as in the interaction while debugging programs.

In the following we will give some examples of programs and program assemblies utilizing features described in this manual. We will simultaneously try to uncover traps which the user may easily fall into to.

This chapter does not describe the various services the user may obtain from the SINTRAN III system such as the Monitor Calls, the Batch Processor, the Real Time (RT) feature, etc. These are described in detail in the documentation: SINTRAN III User's Guide (ND.60.050).

5.1 *LOGGING IN*

Although MAC may be used as a stand-alone program in the NORD-10, it comes naturally in this context to describe MAC and other processors as subsystems running under an operating system, the NORD SINTRAN III Real Time, timesharing and multi-batch system.

Suppose you are already a legal user of the system, just follow these steps:

1. Turn on the terminal. Turn on the on-line switch.
2. Press "Escape" (the ESC key).
3. The terminal responds with the time of day, the date, and the word ENTER.
4. Type the user name, followed by carriage return.
5. The terminal responds by printing PASSWORD.
6. Type your password. If you have none type carriage return. Remember that the password will not be echoed on your terminal.
7. The terminal responds with OK.
8. If the accounting system is active the terminal will print PROJECT-NUMBER. Answer this by typing a project number (a decimal number), followed by carriage return.
9. The terminal prints the character @. This means that it is expecting a command.
10. Any subsystem may be loaded by typing its respective name, for example:

    @MAC
    -MAC-

5.2 *PREPARING A PROGRAM FOR ASSEMBLY*

We have already described in Section 3.2.2.4 the conventional format for programs written in the MAC language. Once written, the program should be typed and transferred to a file using the QED text-editor which is introduced in the following lines:

QED is a powerful text-editing program for use with the NORD COMPUTERS.

It is primarily designed for maintaining source-program-files of multiple languages, such as MAC, NORD-PL, FORTRAN, or BASIC, though it's convenience and ease of usage makes it suitable for all kinds of text-editing.

The text being edited may be read from and written to any mass-storage file or I/O device and text lines may be added, modified, replaced and deleted by a few easy-to-learn commands.

Lines of texts may be addressed in several ways to make it easy for the user to position a specific line or a collection of lines where editing is to be performed. Positioning to a particular line may be specified in the commands themselves, however, just a line address itself can be a command to cause positioning to occur.

A normal editing sequence consists of:

- a READ command to get the old text from a file, or an APPEND if there are only new lines from the terminal.
- APPENDING/INSERTING, CHANGING and DELETING text. LIST specific parts and EDIT single lines to correct errors.
- a WRITE command to save the new text-buffer on a file.
- a FINISH command to leave QED and return to the operating system.

Remember to save your edited text, or it will be lost when a new program is started.

The default tabulator positions correspond to the conventional format of MAC programs. Spaces are normally not written by QED in order to save space in the files, however, MAC in its turn expands tabulator characters to obtain readable assembly listings.

Moreover, form-feeds may be used to divide the program into "pages". Pagination is recommended since it increases program readability. Form-feeds are ignored by MAC. Blank lines may also be included in programs. This also helps readability.

5.3 *ASSEMBLY OF A PROGRAM*

Once MAC is loaded and running, a program may be assembled. If the program to be assembled does not set the location counter before any instructions or constants, set the location counter from the terminal using the slash (/) command.

If you are certain that your program ends with a)LINE command (which terinates assembly and gives control to the terminal), start the assembly process by typing the command)9ASSM with actual parameters.

If while the assembly is in progress, one of MAC's tables fills up, MAC prints out an error message and goes on-line. When this happens, increase the capacity of the tables by using the)9TABL command (which also implies a)CLEAR). Then reset the location counter and start again with)9ASSM.

When the assembly is complete, type a question mark (?) on the terminal to get a list of undefined symbols. Define these symbols either using the "=" or ",", commands on-line or by correcting the program and reassembling it.

Once a program has been assembled, it is often convenient to dump it before starting to have a copy of the original for later debugging. Very often programs are self-destroying due to bugs. The dump interval is set by using the < command and desired dump file is connected to the object stream by using the)9ASSM command. The)BPUN command outputs the absolute binary dump to the object stream and the)LIST command outputs the symbol list to the same stream.

In the following, we will use as an example, a program that

1. outputs a question mark on the terminal
2. reads a file name from the terminal
3. opens the file
4. copies the file to the terminal
5. exits when it encounters the end-of-file character (027)

We assume this program is already on a file written by QED. The file has the name EXAMPLE. This file should be written in an orderly and readable manner.

1. Log in as described in Section 5.1.

2. Type MAC, and MAC prints "—MAC—", when started first time, being ready to accept input from the terminal.

3. Type *: and MAC responds by printing 000001 which is the value of the and MAC responds by printing the contents of location 10. (Later debugging by setting breakpoints does not allow the start address to be 1).

4. Type carriage return and MAC responds by printing line feed as always in one-line mode. A carriage return alone does not assemble anything into location 10, and the location counter remains unchanged.

5. Type)9SCLC which *later* will result in printout of the current location counter preceding the source line on the list stream.

6. Type)9ASSM EXAMPLE, TERM and the following output will appear on the terminal:

```
000010      "88BRF
"
000010      PROGRAM LISTFILE
000011      START, SAA # ?; SAT1;
% **** ERROR AT: 000011 **** RANGE EXCEEDED
  MON 2; MON 65; SAX 0
000015      LES,        SAT 1; MON 1; MON 65; LDT BUFFP; SBYT
000022                  AAA—215;
% **** ERROR AT: 000022 **** RANGE EXCEEDED
  JAZ OPEN; AAX 1; JMP LES
000026      OPEN,       LDX BUFFP; SAT 1; LDA (FTYPE
000031                  MON 50; MON 65; STX FILIN
000034      PRINT,
% **** ERROR AT: 000034 **** ILL. MNEMONIC PRINT
  LDT FILIN; MON 1; MON 65
000037                  AAA—27; JAF *2; MON 0; AAA 27
000043                  SAT 1; MON 2; MON 65; JMP PRINT
% ****ERROR AT: 000046 **** RANGE EXCEEDED
000047      FTYPE,      # SY
000050                  # MB
000051      BUFFP,      BUFFR
000052      BUFFR,      0
000053      BUFFR+200/ 0
000253      FILIN,
% **** ERROR AT: 000253 **** POSSIBLE FAULT 000034

% **** ERROR AT: 000253 **** POSSIBLE FAULT 000033
  0
000254      )FILL
% **** ERROR AT: 000254 **** ( ERROR 000030

000255      "88BRF
"
000255      )LINE
**** 000007 DIAGNOSTICS ****
```

7. We observe some error messages and that the conditional assembly sequence between "88BRF and " are not assembled (printed) because the symbol 88BRF (library mark) is not present in the undefined symbol table. The)LINE passed control to the terminal, and MAC is now expecting further input.

8. Type ? and MAC responds by printing the undefined symbols:

   OGRAM TFILE

   We have obviously forgotten a comment sign (%) in the line: PROGRAM LISTFILE.

9. The error messages are now discussed in turn:

   a) A sharp sign (#) is missing and MAC has taken # ?; as a 16 bit value, also resulting in SAT 1 to be assembled into the same location!

   b) The argument ÷215 is not legal and the instruction should be changed to SUB (215.

c) The symbol PRINT is reserved for the MAC command)PRINT. We decide to change it to PRIUT.

d) The value of the reserved symbol PRINT resulted in a displacement overflow. We can change the occurrence of PRINT throughout the program by using the substitute feature in QED.

e) The definition of FILIN results in two POSSIBLE FAULT warnings. These are really errors because FILIN is referred twice in memory reference instructions. Label FILIN must be moved before the definition of the array BUFFR.

f) The addressing range of a literal is exceeded; i.e., the literal is dumped ()FILL) too far from the referencing instruction. The)FILL must be moved before the BUFFR definition.

10. Correct the source and dump the new version on the same file.

11. Repeat the steps 1 to 6, and the following output will appear on the terminal:

```
000010      ''88BRF
''
000010      % PROGRAM LISTFILE
000010      START,    SAA # # ?; SAT 1; MON 2; MON 65; SAX 0
000010      LES,      SAT 1; MON 1; MON 65; LDT BUFFP; SBYT
000022                SUB (215; JAZ OPEN; AAX 1; JMP LES
000026      OPEN,     LDX BUFFP; SAT 1; LDA (FTYPE
000031                MON 50; MON 65; STX FILIN
000034      PRIUT,    LDT FILIN; MON 1; MON 65
000037                AAA—27; JAF *2; MON 0; AAA 27
000043                SAT 1; MON 2; MON 65; JMP PRIUT
000047      FTYPE,    # SY
000050                # MB
000051      FILIN,    0
000052      )FILL
000054      BUFFP,    BUFFR
000055      BUFFR,    0
000056      BUFFR + 200/ 0
000256      ''88BRF
''
000256      )LINE

**** 000000 DIAGNOSTICS ****
```

12. There are no error messages. Type ? and MAC responds by printing carriage return/line feed. There are no undefined symbols, i.e., the assembly seems to be correct.

13. Type START! or 10! and the program starts printing a ? on the terminal. Everything is OK so far!

14. Type the name of the file to be listed, for instance: EXAMPLE followed by carriage return.

15. The error message NOT OPENED FOR SEQUENTIAL READ indicates that something is wrong with the program. Control is given to the operating system.

The debugging process is described in the next section.

5.4 *DEBUGGING A PROGRAM*

Once a program has been assembled or loaded into memory the program may be run. To start program execution, type an expression to be evaluated to a starting address followed by an exclamation mark (!). For all but the very luckiest of us, this last step will lead to a catastrophic program failure and the program will have to be debugged.

First, get control back to MAC by restarting with the @ CONT command. MAC will now print its characteristic carriage return and line feed. If it does not, the "buggy" program destroyed MAC and MAC and the program must be reloaded (this is an ideal excuse for quitting for the day!).

Once control has been returned to MAC, memory locations can be examined by typing a location number, a symbol or expression followed by a slash (/), e.g.,

200/123456

To change the contents of a memory location, first examine its contents as alone and then type a new value followed by a carriage return. To obtain the value of a symbol, type the value followed by a colon, e.g.,

FOO:000012
*:000400

Numerous other commands useful for debugging are available. One special symbol already mentioned is also very useful during on-line debugging. This is ↑ which has its value as the value of the memory location pointed to by the current location counter. Its use is shown below:

A/ BSET 70 DX ↑ ONE

this changes the instruction in location A to

BSET ONE 70 DX

We will now continue the example from the last section.

16. The @ STATUS command in SINTRAN III is always a good debugging aid, even if the program was interrupted by an ESC. Type STATUS and the following output is printed on the terminal:

    P = 37
    X = 55
    T = 55
    A = 124
    D = 0
    L = 0
    S = 100
    B = 0

    The MON 65 in location 36 has stopped the program, so obviously the OPEN call is correct. From the register dump we see that T = X = 55. Something is perhaps wrong with the T register in MON 1 of location 35?

17. Type CONTINUE to restart MAC.

18. Type PRIUT. and MAC responds with a carriage return/line feed. A breakpoint is now specified in location 34, after the OPEN call.

19. Type START! and the program prints a ?.

20. Type EXAMPLE and control is immediately given to MAC responding by printing . indicating that the breakpoint is reached.

21. Type BT/ and BX/ and BA/ to examine the contents of the respective registers:

    BT/000001
    BX/000055
    BA/000101

    Type FILIN/ to examine the contents of location 51:

    FILIN/000055

    The X register has the value of BUFFR and is not changed through the OPEN call. Obviously, the last instruction executed should be STA FILIN instead of STX FILIN. We can verify this by changing the contents of FILIN and continue execution.

22. Type 101 and carriage return which assembles this new value into FILIN.

23. Type 1! to move the breakpoint to the next location and execute the previous instruction. Type BT/ and MAC prints the contents of the T register BT/000101 which is the new content of FILIN.

24. Type ! to continue execution without more breakpoints and hope.

    The program executes correctly!

25. Correct the last bug by changing the STX FILIN to STA FILIN in the source program.

    While editing, take the opportunity to insert more comments. It is also recommended to change the monitor call numbers to symbols which increases readability and reduces futural maintenance costs.

The final assembly and dump procedure is described in the next section.

5.5 *DUMPING A PROGRAM*

The importance of dumping large programs in the debugging phase is already emphasized. However, any debugged program should always be dumped in a binary version suitable for later retrieval. MAC/SINTRAN III dump and load procedures are described below with some comments.

— BRF Output

Generated by the)9BEG and)9END commands. Can later be loaded by a BRF loader subsystem anywhere in memory. Linking to other BRF program units is also possible.

— Absolute Binary Output

Generated by the)BPUN command. Can later be loaded by)9READ, or the @PLACE-BINARY command in SINTRAN III.

— PROG Files

Generated by the @DUMP command in SINTRAN III. Can later be retrieved by @RECOVER and is the standard format of subsystems. When debugging large programs it is often useful to dump all memory including MAC and its tables.

— DUMP-REENTRANT

Requires an absolute binary file. Later retrieval will utilize the reentrant facility of SINTRAN III.

The following steps continue with the example from the last section and show two ways of assembling, dumping and retrieving the program:

26. Absolute Assembly:

```
@MAC
—MAC—
)9ASSM EXAMP
**** 000000 DIAGNOSTICS ****
1< *
)9ASSM 1,0 ''LISTFILE:BPUN''
)BPUN START
)9EXIT
@PLACE-BINARY LISTFILE:BPUN
@GO  1
```

The program starts execution in location 1.

27. BRF assembly:

```
@MAC
—MAC—
)9SCLC
88BRF
)9ASSM EXAMP, L-P, ''LISTFILE''
**** 000000 DIAGNOSTICS ****
)9EXIT
@NRL
—NORD RELOCATING LOADER—
*SET-LOAD-ADDRESS 20000
*LOAD LISTFILE
*RUN
```

The program starts execution in location 20000.

The listing produced by the)9ASSM EXAMP, L-P, "LISTFILE"" looks like this:

```
000002          % GLOBAL DEFINITIONS
000002          FINIT = 0; INBT = 1; OUTBT = 2; OPENF = 50; ERROR = 65
000002          ''88BRF
000002          )9BEG START
000001          ''
000001          % PROGRAM LISTFILE
000001          % READ FILE NAME FROM TERMINAL (TERMINATOR = 215)
000001          %OPEN FILE FOR SEQUENTIAL READ(DEFAULT TYPE = SYMB)
000001          % PRINT ITS CONTENTS ON THE TERMINAL (TERMINATOR = 27)
000001          START,      SAA # #?; SAT 1; MON OUTBT; MON ERROR; SAX 0
000006          LES,        SAT 1; MON INBT; MON ERROR; LDT BUFFP; SBYT
000013                      SUB (215; JAZ OPEN; AAX 1; JMP LES
000017          OPEN,       LDX BUFFP; SAT 1; LDA (FTYPE
000022                      MON OPENF; MON ERROR; STA FILIN
000025          PRIUT,      LDT FILIN; MON INBT; MON ERROR
000030                      AAA —27; JAF *2; MON FINIT; AAA 27
000034                      SAT 1; MON OUTBT; MON ERROR; JMP PRIUT
000040          FTYPE,      # SY
000041                      # MB
000042          FILIN,      0
000043          )FILL
000045          BUFFP,      BUFFR
000046          BUFFR,      0
000047
000247          BUFFR + 200/ 0
000247          ''8BRF
000247          )9END
000247          ''
000247          )LINE
```

6 INTRODUCTION TO SUBROUTINES

If the same algorithm is to be applied at several different places in a program, it is convenient to put the algorithm's instructions in a subroutine. This subroutine may be called from anywhere where we wish to execute the special algorithm.

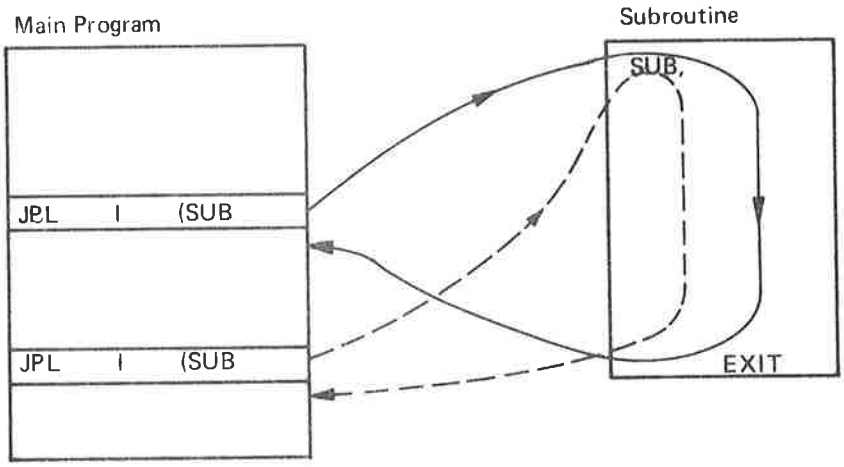


Figure 6.1: Calling a Subroutine

For example, let us assume that the first call of the subroutine SUB in Figure 6.1 is in location 1000 and the next one in location 2000. The subroutine starts, for example, in location 5000. Then the following takes place:

At the first call the P register has the value 1001. The JPL instruction means that the value of the P regiser is copied into the L register and the P register gets the value of the start address of the called subroutine,i.e.,

P = 1001 ⟶ L
SUB = 5000 ⟶ P.

Execution then continues at location 5000, and the subroutine is executed.

Finally, the EXIT instruction in the subroutine is executed. EXIT is equivalent to copying the L register into the P register, i.e.,

L = 1001 ⟶ P.

This actually means a return to the calling program.

At the second call the value of the P register is 2001 which is put into the L register and the P register is again changed to 5000. By executing the EXIT, the P register receives the value 2001 which was saved in the L register. A return to the calling program is performed.

Next, let us assume that the called subroutine also calls a subroutine, as shown in Figure 6.2.

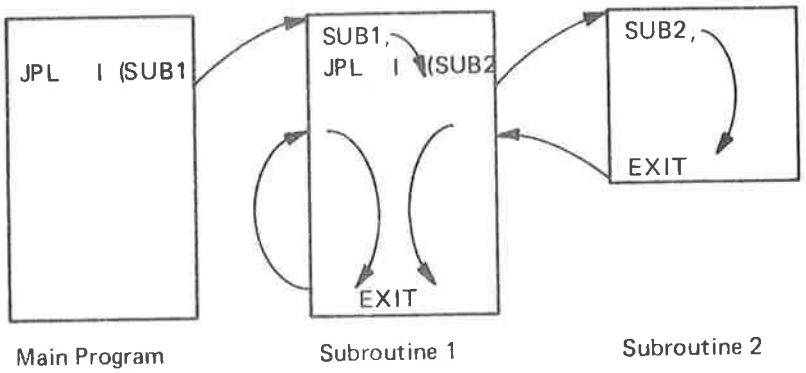


Figure 6.2: Nested Calling of Subroutines

Figure 6.2 shows that control is never returned to the main program. Why? Let us assume that the call in the main program is at location 1000, that SUB1 starts in location 2000 and calls SUB2 in location 2100, and that SUB2 starts in location 3000.

The P and L register then change in the following way:

P = 1001 ⟶ L SUB1 = 2000 ⟶ P	Main program calls SUB1
P = 2101 ⟶ L SUB2 = 3000 ⟶ P	SUB1 calls SUB2
L = 2101 ⟶ P	EXIT in SUB2
L = 2101 ⟶ P	EXIT in SUB1

The last part of SUB1 is repeated infinitely.

Each time a subroutine is called, the L register is changed. This means that a subroutine has to save the L register before it calls another subroutine.

Generally, it is a good habit to save all registers which are to be used in the subroutine before using them, and load them with their original values before returning to the calling program.

```
SUBR,   STF     TAD     % SAVE TAD REGISTERS
        STX     SAVEX   % SAVE X REGISTER
        COPY    SB DA   % SAVE
        STA     SAVEB   % B REGISTER
        COPY    SL DA   % SAVE
        STA     SAVEL   % L REGISTER
        :
        :
        LDA     SAVEL   % LOAD
        COPY    SA DL   % L REGISTER
        LDA     SAVEB   % LOAD
        COPY    SA DB   % B REGISTER
        LDX     SAVEX   % LOAD X REGISTER
        LDF     TAD     % LOAD TAD REGISTERS
        EXIT
TAD,    0;0;0
SAVEX,  0
SAVEB   0
SAVEL,  0
```

6.1 *PARAMETERS*

Usually, it is necessary to send information to and from a subroutine. This is done by the use of parameters. Parameters may be transferred in one of the following three ways:

1. using registers
2. using the locations following the call in the main program
3. using the A register which contains the address of a list of parameter addresses.

These three methods are discussed in the following sections.

6.1.1 *Parameter Transfer via Registers*

If we only want to transfer a small number of parameters to or from the subroutine, we may use the registers. Then the registers which are to transport input parameters, are loaded before calling the subroutine. Output parameters are put into the registers before leaving the subroutine.

Suppose there are four parameters which shall be sent to the subroutine. We may, for example, use the T, A, D and X registers for that purpose.

Let us also suppose that there is only one output parameter which is transferred by the A register.

Main Program:

```
          :
          :
          LDF     P1        % LOAD FIRST 3 PARAMETERS
          LDX     P4        % LOAD FOURTH PARAMETER
          JPL  I  (SUBR
          STA     RES       % STORE RESULT
          :
          :
P1,       ...
P2,       ...
P3,       ...
P4,       ...
RES,      0
          )FILL
```

Subroutine:

```
SUBR,   STF     TAD
        STX     SAVEX
        :
        :
        LDA     OUTPT
        EXIT
TAD,    0
OUTPT,  0
SAVED,  0
SAVEX,  0
```

This method is, of course, only useful if the number of parameters is small.

6.1.1.1 Example

Write a subroutine which computes the sum, the difference and the product of two numbers contained in the A and the D register. The results are to be placed into the T (sum), the A (difference) and the D (product) registers.

Main Program:

```
        :
        :
        LDD     NUM1
        JPL  I  (SUBR
        STF     SUM
        :
        :
        :
NUM1,   ...
NUM2,   ...
SUM,    0
DIFF,   0
PROD,   0
        )FILL
```

Subroutine:

```
SUBR,   COPY    SA DT
        RADD    SD DT     % SUM
        STA     SAVEA     % SAVE A REGISTER
        SWAP    SA DD     % EXCHANGE A AND D
        RSUB    SA DD     % D = D – A
        MPY     SAVEA     % A = A * SAVEA
        SWAP    SA DD     % A = DIFF
                          % D = PROD
        EXIT
SAVEA,  0
```

6.1.2 *Parameter Transfer Via Locations Following the Call*

If the number of parameters get big or are variable, it is convenient to place them into some known locations in the memory. These locations may immediately follow the call. The actual values of the parameters are transferred via these locations.

Main Program:

```
            :
            :
            JPL   I   (SUBR
P1,          <value>
P2,          <value>
            :
            :
PN,          <value>
            :
            :
            )FILL
```

How is it possible to access these parameters from the subroutine? We know that the L register contains the adress of the location following the call. This means then that L = address (P1) = P1. Also, we have to know how many parameters we use in order to be able to calculate the correct return address: L = L + number of parameters. If this number is constant, the L register is incremented by a constant. If the number of parameters is variable, we transfer it as the first parameter and increment the L register by it.

Subroutine with a fixed number of parameters, say n, $1 \leq n \leq 127$:

```
SUBR,    STA       SAVEA     % SAVE A REGISTER
         COPY      SB DA
         STA       SAVEB     % SAVE B REGISTER
         STX       SAVEX     % SAVE X REGISTER
         COPY      SL DB     % FIRST PARAMETER ADDRESS
         SAX       n         % NO. OF PARAMETERS
         RADD      SX DL     % RETURN ADDRESS
         COPY      CM1 AD1   SX DX     % X = −X
NEXT,    LDA ,B ,X n         % LOAD PARAMETER
         :
         :
         JNC       NEXT      % INCREMENT X REGISTER
         LDX       SAVEX
         LDA       SAVEB
         COPY      SA DB
         LDA       SAVEA
         LDA       SAVEA
         EXIT                % RETURN
SAVEA,   0
SAVEB,   0
SAVEX,   0
```

The X register is loaded with the number of parameters. After having negated it, we are able to access consecutive parameters only by incrementing the X register.

Subroutine with a variable number of parameters contained in the first location following the call:

```
SUBR,   STA     SAVEA   % SAVE A REGISTER
        COPY    SB DA
        STA     SAVEB   % SAVE B REGISTER
        COPY    SL DB   % ADDRESS TO
        LDA     ,B      % NO. OF PARAMETERS
        RADD    SA DL   % RETURN ADDRESS
        RINC    DB
NEXT,   LDA     ,B      % ACCESS PARAMETER
        :
        :
        RINC    DB      % INCREMENT B REGISTER
        SKP DB  EQL SL  % TEST IF FINISHED
        JMP     NEXT    % NEXT PARAMETER
        LDA     SAVEB
        COPY    SA DB
        LDA     SAVEA
        EXIT
SAVEA,  0
SAVEB,  0
```

In this, the number of paramaters is also included in the location which actually contains this number.

6.1.2.1 Example

Write a subroutine which computes the sum, the difference and the product of two numbers contained in the first two locations following the call. The results are to be placed in the next three locations following the call in the main program.

Main Program:

```
        :
        :
        JPL   I     (SUBR
NUM1,   ...
NUM2,   ...
SUM,    0
DIFF,   0
PROD,   0
        :
        :
        )FILL
```

Subroutine:

```
SUBR,   STA     SAVEA
        COPY    SB DA
        STA     SAVEB
        COPY    SL DB
        SAA     5           % NO. OF PARAMETERS IS 5
        RADD    SA DL       % RETURN ADDRESS
        LDA     ,B
        ADD     ,B 1
        STA     ,B 2
        LDA     ,B
        SUB     ,B 1
        STA     ,B 3
        LDA     ,B
        MPY     ,B 1
        STA     ,B 4
        LDA     SAVEB
        COPY    SA DB
        LDA     SAVEA
        EXIT
SAVEA,  0
SAVEB,  0
```

6.1.2.2 Example

Write a subroutine which adds a number of numbers. The result is put into the A register. The subroutine uses the following parameters placed in locations following the call:

RADDR	— return address
N	— number of numbers
NUM1	— first number in a field containing at least N numbers

Example:

Main Program:

```
        :
        :
        JPL   I     (SUBR
        CONT
        25
NUM1,   ...
        ...
        :
        :
CONT,   STA         RES
        :
        :
        )FILL
```

Subroutine:

```
SUBR,   STX     SAVEX
        COPY    SB DA
        STA     SAVEB
        COPY    SL DB
        LDA     ,B          % A    CONT
        COPY    SA DL       % RETURN ADDRESS
        RINC    DB          % ADDRESS TO NO. PARAMETERS
        LDX     ,B          % NO. NUMBERS TO BE ADDED
        COPY    CM1 AD1     SX DX
        SAA     0           % CLEAR A REGISTER
        RINC    DB          % INCREMENT B REGISTER
        ADD     ,B
        JNC     *—2
        LDX     SAVEB
        COPY    SX DB
        LDX     SAVEX
        EXIT
SAVEB,  0
SAVEX,  0
```

6.1.3 *Parameter Transfer by Means of the A Register*

The last method we shall discuss is to transfer parameters by only passing the address of a list of parameter addresses to the subroutine via a register, the A register.

This is the way parameters are transferred by FORTRAN or SINTRAN.

Before the main program calls a subroutine, it loads the A register by this address. The list of parameters may be placed anywhere in the memory, and the actual values may even be scattered in memory. But in this case, only executable instructions should follow the call.

Main Program:

```
P1,     ...
Pn,     ...
        :
        :
        LDA     (LIST       % LIST ADDRESS
        JPL     I (SUBR
        :
        :
        MON
        )FILL
P2,     ...
        :
        :
LIST,   P1                  % LIST OF ADDRESSES
        P2                  % TO PARAMETERS
        :
        :
        Pn
```

The only parameter which is directly transferred to the subroutine is contained in the A register. We copy the A register into the B register and are now able to access parameters indirectly through the B register.

Subroutine:

```
SUBR    SWAP    SA DB
        STA     SAVEB     % SAVE B REGISTER
        LDA   I ,B 0      % ACCESS 1st PARAMETER
        :
        :
        LDA   I ,B 1      % ACCESS 2nd PARAMATER
        :
        :
        LDA   I ,B n      % ACCESS nth PARAMETER
        :
        :
        LDA     SAVEB
        COPY    SA DB
        EXIT
SAVEB,  0
```

If we write a MAC subroutine which is called by a FORTRAN program we must not change the B regiser or locations within the B field, i.e., B-200$_8$ through B +177$_8$. In this case it is absolutely necessary to save the B register at the beginning and load it with its original value at the end of the subroutine.

6.1.3.1 Example

Write a subroutine which adds a number of numbers.

The subroutine uses the following parameters:

N	number of numbers
FIELD	start address of the field containing the numbers to be added
SUM	result

Main Program:

```
FIELD   ...
*+n/    ...
        :
        :
        LDA     (LIST
        JPL   I (SUBR
        :
        :
        MON
N,      n
        :
        :
SUM,    0
LIST,   N
        FIELD
        SUM
        )FILL
```

Subroutine:

```
SUBR,   SWAP     SA DB      % B ← LIST
        STA      SAVEB
        STX      SAVEX
        LDX   I  ,B         % X ← NO. OF NUMBERS
        SAA      0          % CLEAR A REGISTER
NEXT,   RDCR     DX         % DECREMENT X REGISTER
        SKP      DX GRE     % TEST IF FINISHED
        JMP      FIN
        ADD   I  ,B ,X  1   % ADD NUMBER
        JMP      NEXT
FIN,    STA   I  ,B   2     % SAVE RESULT
        LDA      SAVEB
        SWAP     SA DB
        LDX      SAVEX
        EXIT
SAVEB,  0
SAVEX,  0
```

APPENDIXES

APPENDIX A

ERROR MESSAGES AND WARNINGS

Messages from SINTRAN III or FILE SYSTEM monitor calls are printed as a self-explanatory text.

MAC messages are printed on the terminal if the list stream is connected to the dummy device. A message is always preceded by the text:

**** ERROR AT: &&&&&& ****

where &&&&&& is a six-digit octal number representing the current location counter. The number of diagnostics is always printed on the terminal when a)LINE or @ is encountered.

The error messages or warnings are listed below in alphabetical order together with an explanation and eventually action to take.

(ERROR

Literal dumped too far from referencing instruction. Insert a)FILL command in the source program within the relative addressing range of the instruction. Reassemble.

)FILL MISSING

Insert a)FILL command before the)9END in the source program. Reassemble.

ALREADY DEFINED <symbol>

The symbol indicated has already been defined in the local symbol table. This is not necessarily an error. The value of the symbols latest definition is used.

BREAKPOINT NOT RESTORED

This warning indicates that a debugging run was started (!) without reaching any breakpoint. However, all breakpoints specified are intact.

CHECKSUM ERROR

A checksum error was detected in a)9READ command. Try again or generate a new binary file.

DISASSEMBLER ERROR

Irrecoverable error in the disassembler option.

ENT DEFINED

A symbol given in a)9ENT command was previously defined. Reassemble deleting either the symbol's previous definition (probably *before* the)9BEG command) or its inclusion in the)9ENT command.

EXT DEFINED

A symbol given in a)9EXT command was previously defined. Reassemble deleting either the symbol's previous definition or its inclusion in the)9EXT command.

EXT IN ADDRESS ARITHMETIC

An arithmetic expression included an external symbol. For example,

```
)9BEG
)9EXT     PER
          LDA     I (PER + 1   % ILLEGAL
```

Correct in the source program and reassemble.

EXT MISSING

An external symbol was included in a)KILL command. For example,

```
)9EXT PER
JPL     I   (PER
)KILL PER       % ILLEGAL
)FILL
```

Delete the symbol from the)KILL command in source program and reassemble.

ILL. ADDRESS

An attempt was made to assemble or jump (via the) command) into MAC itself. If the latter, try again with a legal address. If the former, correct the program and reassemble.

ILL. BRF UNIT INITITIATION

A)9ENT or)9EXT was used somewhere other than before the first instruction or constant after a)9BEG, or)9BEG was used doubly. Fix the program and reassemble.

ILL. BREAKPOINT

Several conditions may cause this message:

— illegal start or brek address
— redefinition of a breakpoint
— maximum number of breakpoints set
— undefined symbol in address expression preceding ! or .

ILL. CHARACTER

An illegal character was found in the source stream. The character is ignored.

ILL. EXPRESSION

MAC has encountered an expression having double relocation or direct access to an external symbol. For example,

```
)9BEG
)9EXT     PER
A,        0
B,        0
B–A                          % LEGAL
B+A                          % ILLEGAL
          LDA      PER       % ILLEGAL
          LDA      I (PER    % LEGAL
```

Correct the source program and reassemble.

ILL. INSTRUCTION

Something is illegal about an instruction, for instance, JAZ ,B. Correct source program and reassemble.

ILL. MNEMONIC

An attempt was made to redefine one of MAC's built-in symbols in the main symbol table. The attempted definition is ignored.

ILL. USE OF COMMAND

Illegal use of a command. For example,

)KILL 5

Fix the source program and reassemble.

MACRO ERROR NO 000001

An attempt was made to redefine a macro or name conflict with another symbol. The macro definition is ignored.

MACRO ERROR NO 000002

The macro tables overflowed. The macro definition is ignored.

MACRO ERROR NO 000003
MACRO ERROR NO 000004

Both these error messages indicate the use of a symbol preceded by a $ within a macro body but not declared in the macro's formal parameter list. The macro definition is ignored.

MISSING PARAMETER

A macro call has insufficient parameters. The call is ignored.

OPTION MISSING

An option was "called" which was not included in MAC. Either do not attempt to use the option or construct a version of MAC including the option. Reassemble.

POSSIBLE FAULT

There was possibly, but not necessarily, a fault in assembly into the indicated location. This message may occur when a symbol is defined and references in the undefined symbol table are updated. The following conditions are checked:

- if not in BRF assembly mode, and
- if the content of the refrence address is different from zero, and
- if the address range (–200) is exceeded
- then the message is given due to the fact that MAC has "forgotten" whether the symbol appeared in an instruction or in an address expression.

Use the two-pass assembly option, or include the commands related to BRF in conditional assembly. Then examine the location and correct if necessary.

RANGE EXCEEDED

An attempt was made to reference a location outside the addressing range of the referencing instruction, or an argument was outside its limits in an argument instruction. Fix the source program and reassemble.

TABLE FULL / <table>

One of MAC's tables overflowed. Use)9TABL to expand the table(s).

SYMBOL NOT DEFINED <symbol>

A symbol included in a command was *not* previously defined. Define the symbol and try again. If you are not lucky, reassemble.

UDEF ENTRY

An entry was still not defined at)9END. This doesn't necessarily mean it's an error, but if it is add the appropriate)9ENT command to the source program and reassemble.

WHAT?

Illegal use of the) command. Give the correct command on-line and continue assembly.

APPENDIX B

BUILT-IN SYMBOLS

B.1 *MAIN SYMBOL TABLE (Instruction Mnemonics and Commands)*

CLEAR	9ASCI	9LITR	9PARI	9TSS	9EXIT
9ASSM	9BEG	9END	9EXT	END	9EOF
9ADS	9ASF	9LC	9RT	9MSG	9MSGE
9LIB	9FABS	9SET	CLD	WMNE	WLOC
BPUN	KILL	WRITE	WRTM	NWRT	PRINT
PUNCH	LINE	WRUS	FILL	SSK	SSZ
SSQ	SSO	SSC	SSM	ALD	EXR
IRR	IRW	IDENT	IIC	IIE	IOX
LBYT	SBYT	LMP	LRB	SRB	MGRE
MLST	GEQ	LSS	MON	MIX3	PEA
PES	PGS	PL10	PL11	PL12	PL13
PCR	PON	POF	PVL	RDIV	RMPY
SSTG	ION	IOF	2OR3	TRA	OPR
STS	PID	PIE	TRR	MCL	MST
WAIT	BCM	BAC	BSTA	BSTC	BLDA
BLDC	BANC	BORC	BAND	BORA	RAND
RORA	REXO	CM2	CM1	RCLR	RINC
RDCR	AD1	ADC	IF	ROT	ZIN
LIN	SHT	SHD	SHA	SAD	SKP
EQL	UEQ	GRE	LST	EXIT	RSUB
RADD	ONE	ZRO	BSKP	BSET	SWAP
COPY	SD	SP	SB	SL	SA
ST	SX	DD	DP	DB	DL
DA	DT	DX	SHR	I	,B
,X	NLZ	DNZ	SAA	AAA	SAX
AAX	SAT	AAT	SAB	AAB	LDA
STA	JMP	JPL	STT	MPY	STZ
STF	STD	AND	ORA	JAP	JAF
JAN	JAZ	JPC	JNC	JXC	JXN
LDF	FAD	FSB	FMU	FDV	LDD
STR	LDR				

B.2 *LOCAL SYMBOL TABLE ("Optional" Commands)*

9REG	9CON	9TR	9RDS	9DS	9ROT
9BAR	9MAR	9TAR	9LAR	9PPO	9TPO
9STOP	9INN	9OVAL	9NVAL	9FLT	9STEP
9IOT	9DEVN	9PRIV	FIX	9TABL	9READ
9RCLC	9SCLC	9RLPL	9SLPL	LSTM	MCDEF
9DBUG	2PASS	1PASS	RESSM	SETSM	OCT
DEC	BIR	BB	BSTS	BL	BD
BA	BX	BT	BP	MASK	OLD
NEW	CHANGE	LIST	CORE	PCL	ZERO

APPENDIX C

MAC SPECIAL VERSIONS

This special version of MAC is desgined to operate under SINTRAN III. From the user's point of view there is no difference in operating MACF or MAC.

When not in BRF output mode, the output goes to a 64K (maximum) random file called the image-file. This file is expanded during assembly as required by the program size, thus avoiding waste of mass storage space. All the commands in MAC which access memory are in MACF designed to access the image-file. For instance,)ZERO,)PRINT,)BPUN, etc.

The main purpose of MACF is to allow the user to build systems anywhere in memory.

When started MACF requests the name (or number) of the actual image-file. Here are some examples:

1. IMAGE-FILE:

   Carriage return defines the image-file to be the user's standard scratch file (file no. 100).

2. IMAGE-FILE: PER

   The name of an existing file terminated by carriage return. This file is automatically opened. The default file type is CORE. A number may optionally be supplied indicating an already opened file with access code "WX".

3. IMAGE-FILE: "PER"

   The name of a new file enclosed in double quotes (") and terminated by carriage return. This file is created and opened. The default file type is CORE.

If MACF is restarted using the @ CONTINUE command, the image-file name is requested again, unless the standard scratch file is being used.

During assembly MACF outputs absolute binary code to a buffer pool. This pool is exhausted, i.e., written onto the image file by the)LINE.)9EXIT has the same effect, but also closes all open files and returns control to SINTRAN III. Note that the escape key (ESC) is a *dangerous alternative* to)9EXIT because the buffer pool will not be exhausted.

The NEW, OLD and MASK locations belonging to the)CHANGE command must be examined and/or changed by the @LOOK-AT command in SINTRAN III. First use the colon (:) in MACF to examine the value (addresses) of the three symbols. The breakpoint option is not available.

C.1.1 *Additional Commands*

C.1.1.1 The)9MOVE Command

This command is used to move a block of image from one place to another.)9MOVE must be followed by three standard MAC symbols separated by spaces and terminated by carriage return. The value of the symbols are taken to be:

1. a source address
2. a destination address
3. word count

Example:

```
)9 MOVE   A   B   C
```

C.1.1.2 The)SYSDF and)ULIST Commands

)SYSDF puts MACF in a system definition mode. This mode is reset by)LINE (@).

While in system definition mode only those symbols referred to in the undefined symbol table will be defined when inputting an equal sign definition to MACF. All other equal sign definitions are ignored.

The purpose of this mode is to avoid filling up the tables with unnecessary symbols.

)ULIST is associated with the object stream. The command makes it possible to link several separately assembled, but interrelated programs using the assembler.

)ULIST outputs the undefined symbol table in symbolic code with the following format:

```
<octal address> / ↑<undefined symbol name>      % ↑ means previous contents
                                                % in this location
:
@                                               % )LINE
```

Proposed use:

Each program part is separately assembled and)LIST,)ULIST and)BPUN files are produced for each part. Finally, the different parts are linked together by the following three steps:

1. Load all binary files ()9READ)
2. Input all)ULIST files ()9ASSM)
3. Input all)LIST files ()9ASSM)

The system definition mode ()SYSDF) may successfully be used in step 3.

C.2 *MACM (MAC Mass Storage)*

MACM is a modified standard MAC assembler. The main difference is its ability to assemble programs out on a mass storage device (disk or drum) in a memory image format. Later appearance of disk must be considered as *drum* if this is the actual mass storage.

MACM is a stand alone system and when running it has complete control of the CPU and external devices.

MACM has the capability of swapping itself with the memory image on mass storage and start in a specified location. Used together with the CTOM bootstrap program, the user may freely change between the MACM assembler and his own program much the same as with an ordinary MAC assembler. However, the problem of protecting MACM from the user program is non-existant and all memory is available to the user.

"The user program" may, of course, be any program, but in this context it is the SINTRAN Operating System.

As an aid to debugging MACM has been equipped with commands to save ()9STOR) and restore ()9REST) the current image to and from a save area on the disk. The saved area may be compared with the image area word by word by means of the compare command ()9COMP). The image area may be loaded from memory using CTOM after having run the program.

A "system definition mode" ()SYSDF) may be used when linking ()9ASSM) and loading ()9READ) the programs. This mode ensures that only those system-symbol definitions which are referred to will be taken care of, others are ignored, when reading in a list of definitions from a system or main program.

In order to use MACM to link programs separately assembled, the undefined reference list must be generated ()ULIST).

The image or parts of it may be moved to/from the segment area (of SINTRAN) by the commands ()9SAVE,)9GET).

C.2.1 *Special MACM Commands*

)9STOR

copies the complete current memory image to the save area.

)9REST

restores the memory image from the save area.

)9COMP

compares word by word the image area and the save area. Any differences are output to the device connected to the list stream. The)9COMP command is used with lower and upper limits for the comparison:

```
A < B
)9COMP
```

The current state of the registers on all levels and the page tables are printed. Then words from address A to address B, both inclusive, are compared.

)SYSDF

works as described in Appendix C.1.

)ULIST

works as described in Appendix C.1.

)LINE

In addition to the previous definition of this command, the following will be effectuated:

1. reset system definition mode
2. update memory image by emptying the mass storage block buffer in MACM

)9CTOM

is connected to the object stream. It produces two octal bootstraps described in Section C.2.2. It is important to remember that some parameters given to)9BYTT or)9ALTR are used. Thus, a CTOM bootstrap always corresponds to the latest)9BYTT command.

)CTOM1

generates a bootstrap on the floppy disk, unit 0, which is equal to the first part of the CTOM sequence produced by the)9CTOM command (memory to image).

)CTOM2

generates a bootstrap on the floppy disk, unit 0, which is equal to the second part of the CTOM sequence produced by the)9CTOM command (MACM to memory).

)9SBLO <number>

is used to manipulate the current block being read or written by MACM through the stand alone I/O system (IOXLIB). The current block is written with)9SBLO 177777 (if sequential output to floppy disk or magnetic tape).

Note! This block must not be confused with the mass storage block buffer which is a part of MACM.

)9BYTT <10 symbols separated by space>

This command makes it possible to change the "basic parameters" of a MACM system. This means, for example, that the same binary version of MACM may be used for drum as well as different disks. The ten symbolic parameters for)9BYTT must be previously defined. The meaning of the parameters is explained below:

MSTYP:	Mass storage type (described later)
DEVNO:	Primary mass storage device number
CORAD:	Start address of core load in memory*
LONG:	Length of core load in words*
CLM:	Upper limit for core load numbers (inclusive)*
BLST:	Mass storage address of the segment area
DRES:	Mass storage address of memory image
CRMAX:	End of memory address (77777 for 32k memory)
MACAD:	Mass storage address of area where MACM is saved
DASA:	Mass storage address of save area

* Must be specified but are dummy for SINTRAN III.

The symbol names may of course be anything, but the order of the parameters is essential (as described).

After the symbols have been given the desired values, type the command:

)9BYTT MSTYP DEVNO CORAD LONG CLM BLST DRES CRMAX MACAD DASA

MACM now writes carriage return/line feed indicating that the command has been executed. If a symbol in the parameter string is not defined, the error message:

SYMBOL NOT DEFINED <symbol>

is printed. Restart with)9BYTT.

)9ALTR <up to 10 symbols or numbers separated by commas>

This command is much like)9BYTT, but only specific parameters may be altered. The order of the parameters are the same as with)9BYTT. Parameters will remain unchanged if skipped by typing commas or carriage return. Numbers may be used instead of defined symbols.

Examples:

)9ALTR ,1550

changes only the primary mass storage device number (DEVNO).

)9ALTR ,,,,,,77777

changes only the upper memory address (CRMAX).

```
)9SAVE  A  B  C        % IMAGE TO DISK
)9GET   A  B  C        % DISK TO IMAGE
```

A number of 1k blocks is moved to/from memory image and the segment area.

The three parameters must be defined MAC symbols, and the values are taken to be:

A	= a memory image address
B	= size (1k blocks)
C	= disk address (1k blocks relative to BLST)

C.2.2 *Loading and Running*

Loading

The loading procedure of SINTRAN is described in a separate documentation. Generally, programs are loaded on the image by using the)9READ command. The linking is performed by reading ()9ASSM) the ")LIST" and ")ULIST" information. However, MACM is also able to assemble symbolic information, for instance, patches to SINTRAN.

Starting

Transferring control from MACM to a user program is done the usual way by writing a start address followed by the exclamation mark. This will cause MACM itself to be saved on the disk in a MACM save area, the current image is read into memory and control is transferred to the specified location.

Note: The upper 60 locations of the image should not be used as these are used by the transfer routine.

Using start address 0 (zero) will not cause a transfer to location zero, but rather force a JMP * to be executed in the transfer program when swapping is finished.

Return to MACM

When the user has attempted execution of his program, he may want to return to MACM either to make corrections in his current program or for reload of memory of a new program.

This may be done with a 2-part bootstrap program in octal format read by the microprogram and called CTOM.

The first part is used if the user wants to save his current memory, i.e., memory is written on the image area of the disk.

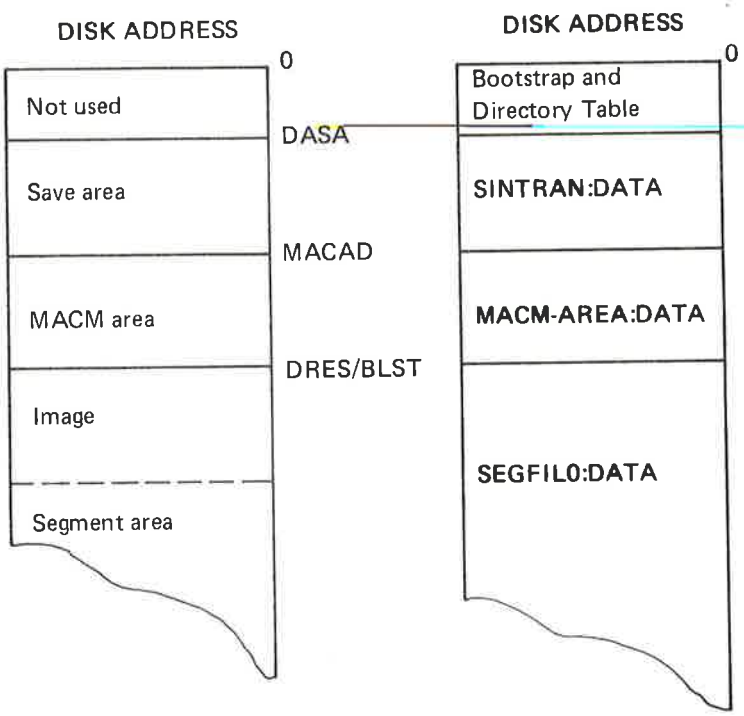
The second part causes MACM to be loaded into memory from its save area and started.

On paper tape the two parts of the CTOM tape are separated by some blank tape so the user may only use the second part if he so desires. However, this will cause his current memory status to be lost.

C.2.3 *Other Information*

Mass Storage Layout

The figure below shows the layout on disk and the relationship MACM/SINTRAN.



Mass Storage Types

The first parameter of the)9BYTT or)9ALTR command is the mass storage type (MSTYP). At present the following mass storage types are available:

0 = drum
1 = dummy
2 = CDC cartridge disk
3 = CDC 33/66 megabyte disk
4 = CDC 38/75 megabyte disk
5 = CDC 288 megabyte disk

Note that MACM always accesses unit 0!

Logical Device Numbers

MACM utilizes the IOXLIB for sequential input/output. At present the following devices are implemented:

Device:	Input:	Output:	Comments:
Dummy	—	0	
Teletype 1	1	1	
Tape-Reader	2	—	
Tape-Punch	—	3	
Card-Reader	4	—	
Line-Printer	—	5	
Link	—	—	
Teletype 2	7	7	
Floppy-Disk	10	10	
Mag.Tape	11	11	Tandberg/Pertec
Mag.Tape	12	12	Hewlet-Packard

Special Conditions

The MACM system, including IOXLIB and CTOM may halt the execution (WAIT). Below is a list of the numbered WAIT instructions which may be executed.

WAIT 0

The first part of the CTOM bootstrap (memory to image) has executed correctly.

WAIT 17

An error condition has occurred when accessing the logical device number 10, 11 or 12. The WAIT 17 is executed after 10 retries.

WAIT 20

An error condition has occurred in the access of the actual mass storage through the CTOM or swapping program. The status register is displayed in register A. Inclusive or of errors is tested, but no retries are performed.

APPENDIX D

ABSTRACTS

D.1 *NORD-10/S INSTRUCTION CODE*

			15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	000.000	STZ	0	0	0	0	0											
	004.000	STA	0	0	0	0	1											
	010.000	STT	0	0	0	1	0											
	014.000	STX	0	0	0	1	1											
1	020.000	STD	0	0	1	0	0											
	024.000	LDD	0	0	1	0	1											
	030.000	STF	0	0	1	1	0											
	034.000	LDF	0	0	1	1	1											
2	040.000	MIN	0	1	0	0	0	XIB				DISPLACEMENT Δ						
	044.000	LDA	0	1	0	0	1											
	050.000	LDT	0	1	0	1	0											
	054.000	LDX	0	1	0	1	1											
3	060.000	ADD	0	1	1	0	0											
	064.000	SUB	0	1	1	0	1											
	070.000	AND	0	1	1	1	0											
	074.000	ORA	0	1	1	1	1											
4	100.000	FAD	1	0	0	0	0											
	104.000	FSB	1	0	0	0	1											
	110.000	FMU	1	0	0	1	0											
	114.000	FDV	1	0	0	1	1											
5	120.000	MPY	1	0	1	0	0											
	124.000	JMP	1	0	1	0	1											
	130.000	CJP	1	0	1	1	0	SUBIN.										
	134.000	JPL	1	0	1	1	1											
6	140.000	SKP + EXT	1	1	0	0	0	SUBIN.				EXT			S		D	
	144.000	ROP	1	1	0	0	1	RAD	ADC	AD1	CM1	CLD						
	150.000	MIS	1	1	0	1	0	SUBIN.										
	154.000	SHT	1	1	0	1	1	ZIN	ROT	SHA	SHD		NUMBER OF SHIFTS					
7	160.000	N.A.	1	1	1	0	0											
	164.000	IOX	1	1	1	0	1	DEVICE ADDRESS										
	170.000	ARG	1	1	1	1	0	FUNCTION				ARGUMENT						
	174.000	BOP	1	1	1	1	1	FUNCTION				BIT NO.			D			
			100.000	40.000	20.000	10.000	4.000	2.000	1.000	400	200	100	40	20	10	4	2	1

D.2 *NORD-10/S ADDRESSING MODES*

The displacement may consist of a number ranging from −128 to +127. Therefore, this addressing mode gives a dynamic range for directly addressing 128 locations backwards and 127 locations forwards.

Generally, a memory reference instruction will have the form:

<operation code> <addressing mode> <displacement>

Note that there is no addition in execution time for relative addressing, pre-indexing, post-addressing or both. Indirect addressing, however, adds one extra memory cycle to the execution time.

The address computation is summarized in the table below. The symbols used are defined as follows:

,X	Bit 10 of the instruction
I	Bit 9 of the instruction
,B	Bit 8 of the instruction
D	Contents of bits 0-7 of the instruction (displacement)
(X)	Contents of the X register
(B)	Contents of the B register
(P)	Contents of the P register
( )	Means contents of the register or word

The effective address is the address of that memory location which is finally accessed after all address modifications (pre- and post-indexing) have taken place in the memory address computation.

Addressing Mode	Effective Address	Mnemonic	X	I	B
P-relative	(P) + D		0	0	0
B-relative	(B) + D	,B	0	0	1
indirect P-relative	((P) + D)	I	0	1	0
indirect B-relative	((B) + D)	,B I	0	1	1
X-relative or indexed	(X) + D	,X	1	0	0
B-relative indexed	(B) + D + (X)	,B ,X	1	0	1
indirect P-relative indexed	((P) + D) + (X)	I ,X	1	1	0
indirect B-relative indexed	((B) + D) + (X)	,B I ,X	1	1	1

D.3 *REGISTER OPERATIONS MEMO*

In the examples below A is used as destination register while source register is B or zero. C is the carry indicator, A and B may be exchanged by any general register, but the user should always be careful when the program counter is involved.

.NOT. is a logical operator (⌐)
.AND. is a logical operator (∧)
.EXOR. is a logical operator (∀)
.INOR. is a logical operator (V)

RCLR DA	% A: = 0
RDCR DA	% A: = A − 1
COPY CM1 DA	% A: = − 1
RINC DA	% A: = A + 1
COPY AD1 DA	% A: = 1
RADD ADC DA	% A: = A + C
COPY ADC DA	% A: = C
RDCR ADC DA	% A: = A + C − 1
COPY ADC CM1 DA	% A: = C − 1
RADD SA DA	% A: = 2*A
COPY CM1 SA DA	% A: = − A − 1
RINC SA DA	% A: = 2*A + 1
RSUB CLD SA DA	% A: = − A
RADD ADC SA DA	% A: = 2*A + C
COPY ADC CM1 SA DA	% A: = − A + C − 1
RADD SB DA	% A: = A + B
RDCR SB DA	% A: = A − B − 1
COPY CM1 SB DA	% A: = − B − 1
RINC SB DA	% A: = A + B + 1
COPY AD1 SB DA	% A: = B + 1
RSUB SB DA	% A: = A − B
RSUB CLD SB DA	% A: = − B
RADD ADC SB DA	% A: = A + B + C
COPY ADC SB DA	% A: = B + C
RDCR ADC SB DA	% A: = A − B + C
COPY ADC CM1 SB DA	% A: = − B + C − 1
REXO CM1 DA	% A: = .NOT. A
SWAP SB DA	% A: = B, B: = A
SWAP CLD SB DA	% A: = B, B: = 0
SWAP CM1 SB DA	% A: = − B − 1, B: = A
SWAP CM1 CLD SB DA	% A: = − B − 1, B: = 0
RAND SB DA	% A: = A .AND. B
RAND CM1 SB DA	% A: = (.NOT. B).AND. A
REXO SB DA	% A: = A .EXOR. B
REXO CM1 SB DA	% A: = (.NOT. B).EXOR. A
REXO CM1 CLD SB DA	% A: = .NOT. B
RORA SB DA	% A: = A .INOR. B
RORA CM1 SB DA	% A: = (.NOT. B).INOR. A

D.4 *ASCII CODES*

APPENDIX E

32 BITS FLOATING POINT

As the NORD-10 may be supplied with a microprogram which operates on 32 bits floating point numbers, a special MAC version is available for users who stick to this format.

The main difference in assembly programs is the load and store operations of the floating accumulator utilizing LDD/STD.

The possibility of maintaining the *same* source programs for 32 or 48 bits floating point hardware is very important. Accordingly, the following feature is implemented in MAC:

Mnemonic	Equals (32)	Equals (48)	Comment
LDR STR 2OR3	LDD (24000) STD (20000) 2	LDF (34000) STF (30000) 3	Load Real Store Real For tables, etc.

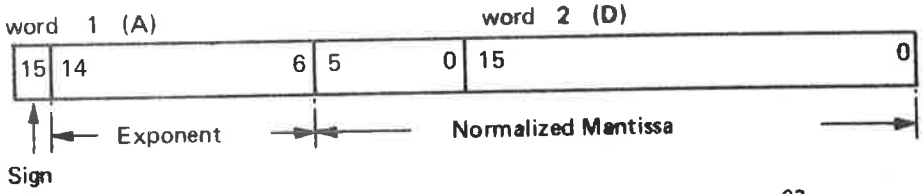
The data format of floating point words is 22 + 1 bits mantissa magnitude, one bit for the sign of the number and 9 bits for a signed exponent.

The mantissa is always normalized, $0.5 \leq$ mantissa < 1. The exponent base is 2. The exponent is biased with 2^8, i.e., 400_8 is added to the actual exponent, so that a standarized floating zero contains zero in all 32 bits.

In the computer memory one floating point data word occupies two 16 bit locations, which are addressed by the address of the exponent part.

n	exponent, sign bit and most significant part of mantissa
n + 1	least significant part of mantissa

In CPU registers, bits 0-15 of the mantissa are in the D register, bits 16-31, the most significant part of the mantissa, exponent and sign, in the A register. These two registers together are defined as the floating accumulator.



The accuracy is 23 bits of 6-7 decimal digits, any integer up to $2^{23} - 1$ has an exact floating point representation.

Note: The one extra bit in the mantissa is the most significant, and is set to one if not all bits in the exponent are zero. It is removed in the result.

The range is

$2^{-256} * 0.5 \leq X < 2^{255} * 1$ or $X = 0$

or

$10^{-76} \quad X \cdot 10^{76}$

Examples (octal format):

	A	D
0:	0	0
+1.0:	040100	0
−1.0:	140100	0
+3.0:	040240	0

The instructions affected are:

FAD	Floating Point Add
FSB	Floating Point Subtract
FMU	Floating Point Multiply
FDV	Floating Point Divide
NLZ	Convert Integer to Floating Point
DNZ	Convert Floating Point to Integer

The normalize and denormalize operations for 32-bits floating point use the same instruction codes as for 48-bits floating point operations, but do not affect the T register. For the 32-bits DNZ operations, the scaling factor should *always* be -20_8 other scaling factors will not cause a different result, but will affect the test for overflow.

INDEX

The index is being compiled and will possibly be distributed within this century.!!!

INDEX

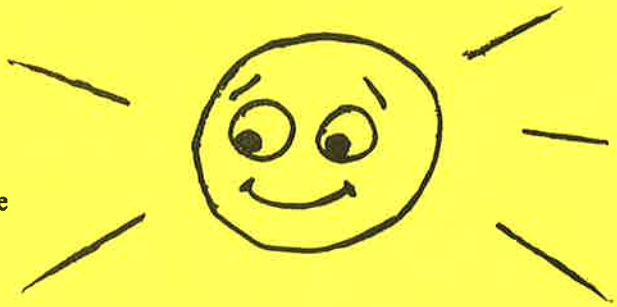
* * * * * * * * * * SEND US YOUR COMMENTS!!! * * * * * * * * * *

Are you frustrated because of unclear information in this manual? Do you have trouble finding things? Why don't you join the Reader's Club and send us a note? You will receive a membership card - and an answer to your comments.

Please let us know if you

* find errors
* cannot understand information
* cannot find information
* find needless information

Do you think we could improve the manual by rearranging the contents? You could also tell us if you like the manual!!



* * * * * * * * HELP YOURSELF BY HELPING US!! * * * * * * * *

Manual name: MAC - Interactive Assembly and Debugging System, User's Guide

Manual number: ND-60.096.01

What problems do you have? (use extra pages if needed) _____

Do you have suggestions for improving this manual? _____

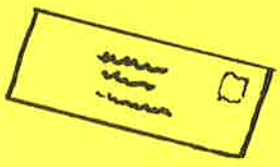
Your name: _____ Date: _____

Company: _____ Position: _____

Address: _____

What are you using this manual for? _____

Send to: Norsk Data A.S.
Documentation Department
P.O. Box 4, Lindeberg Gård
Oslo 10, Norway



Norsk Data's answer will be found on reverse side

Answer from Norsk Data _____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

Answered by _____ _____ _____ _____ Date _____

Norsk Data A.S.

Documentation Department

P.O. Box 4, Lindeberg Gård

Oslo 10, Norway

– we make bits for the future

NORSK DATA A.S BOX 4 LINDEBERG GÅRD OSLO 10 NORWAY PHONE: 30 90 30 TELEX: 18661