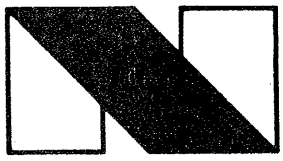
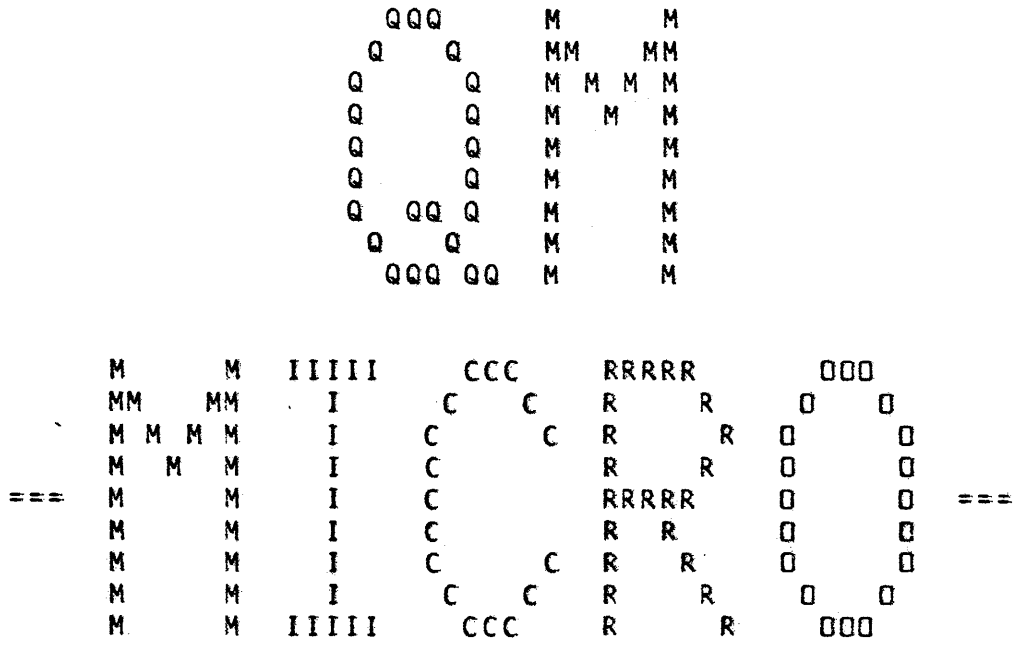
QM
MICRO

QM MICROASSEMBLER
REFERENCE MANUAL



NANODATA CORPORATION
2457 Wehrle Drive Williamsville, New York 14221
(716) 631-5880
San Diego, Calif. 92120
(714) 464-3025



Q M M I C R O A S S E M B L E R

R E F E R E N C E M A N U A L

Version I.3

First edition

NANODATA CORPORATION
2457 Wehrle Drive
Williamsville, New York 14221
716-631-5880

TABLE OF CONTENTS

APPENDICES

I INTRODUCTION

The QM MICROASSEMBLER is a main store program written for the emulated "NOVA" computer to satisfy two assembly language processing needs:

- assembly of QM/SYSTEM control store microprograms.
- assembly of "NOVA" format assembly language programs.

The QM MICROASSEMBLER will be referred to by the name "MICRO" throughout this manual.

This manual provides source language specifications and operating instructions for programmers writing either class of program. NOVA format assembly capability is provided for those maintaining or modifying the NANODATA CONTROL SYSTEM (NCS), which is written utilizing NOVA type instructions. Beyond that, MICRO is extensible and will be restructured, dynamically, to conform to any of the microprogramming architectures defined for execution on the QM/SYSTEMS.

It is assumed that the reader is familiar with the terminology and operation of the NANODATA CONTROL SYSTEM. Additional information on NCS is provided in the QM - NCS OPERATIONS GUIDE.

1.1 OVERVIEW

MICRO is a conventional two pass assembler with special provision for the definition of microinstruction operation codes, formats and constants when used for the assembly of QM/SYSTEM microprograms.

Two NCS files are used as input. They are:

DEF - a binary output file from a QM/SYSTEM Nanoassembly, containing the definition of microinstruction operation codes, formats and constants. This file can be omitted when assembling NOVA programs. Up to four DEF files can be assigned for each assembly.

INPT - a source file containing the source program to be assembled. This file is always required. An additional three files may, optionally, be assigned to INPT as symbolic definition files. These files will not produce any generated object code, but will provide common symbolic values for reference during the assembly.

One NCS file is produced as output. It is:

BIN - a loadable format, binary output file of the assembled object program. This file may be omitted if only an assembly check, or listing, is desired.

An optional printer listing is also produced as a result of an assembly. This listing shows the object code addresses, the assembled object code, and the source statements, along with an indication of any errors detected during the assembly.

An alphabetized summary of all operation codes and all symbolic names may be listed, optionally, at the end of the assembly.

1.2 ASSEMBLER ORGANIZATION

MICRO consists of four basic phases. These are:

1) DEFINITION phase - the DEF file (if any) is processed and a symbol table is constructed for the microinstruction operation codes and constants appropriate to the microprogram environment for which the assembly is to be done. If no DEF file is specified, MICRO assumes that the assembly is to be of a NOVA type program, and built-in symbol tables and formats appropriate to the NOVA architecture are used.

2) PASS 1 - The INPT files are processed and all labels are evaluated and placed into the symbol tables. Memory space is allocated through use of a micro-location-counter, which is incremented each time a memory word is generated. The values assigned to Labels are usually that which is found in the micro-location-counter as Labels are encountered.

3) PASS 2 - the primary (first) INPT file is again processed and the object code corresponding to each statement is generated. If specified, the BIN file is produced during this phase. The output listing is also produced if desired.

4) SUMMARY phase - the symbol tables are scanned to produce an alphabetized listing of the operation codes and symbolic names used in the program.

1.3 UTILIZATION

MICRO is designed to operate under the NANODATA CONTROL SYSTEM, utilizing input / output and loader facilities provided therein. An assembly may be initiated from the system console or from control records previously placed into the system "COMMANDS" file. All general assembly options and file names are specified during initiation.

The resulting binary, object program, file is produced in either QM/SYSTEM microcode format, if any DEF files were specified, or NOVA code format, if no DEF file was declared. NOVA object files are in a format equivalent to that required for loading on standard NOVA systems. QM/SYSTEM object files are provided in an internal format which is capable of being loaded into either of the 18 bit QM/SYSTEM memories, control store or main store. The object file formats are fully described in section 6.2 of this manual.

Assembler operating procedures are described in detail in chapter 7.

2 LANGUAGE STRUCTURE

This section discusses the rules for structuring source program statements. Statements are classified into four types: comment, control, instruction, and data. Where applicable, statements are further subdivided into fields, and these fields are themselves defined in terms of elements.

2.1 STATEMENT CLASSIFICATION

Statements are always contained within one source INPT file record. There is no continuation of any statement on the succeeding record. Each new statement is classified by either the first character or first type of element recognized, when scanning the statement from left to right. The following subsections discuss the meaning of each statement type.

2.1.1 COMMENTS

A comment statement is used to annotate the program listing. Any statement that has an asterisk (*) or semicolon (;) in its first character position will be treated as a comment statement. It will be printed but will have no other effect on the program translation.

A semicolon may appear in any position on any statement, causing all characters to the right to be considered as comments for that statement. Should the first non-blank character on a statement be a semicolon then that statement is also considered a comment statement. The only exception to this rule is that semicolons may be used as character data where text strings are allowed.

2.1.2 CONTROL (PSEUDO OPERATIONS)

MICRO maintains certain symbolic names in a special reserved name list, known as PSEUDO OPERATIONS. When a statement begins with a PSEUDO OPERATION name that statement is classified as a CONTROL statement. Each CONTROL statement has an immediate effect on MICRO, to control the current assembly.

PSEUDO OPERATIONS may be used to control the source statement listing, conditionally determine whether object code should be generated, specify the microprogram location counter, or to generate text string data. PSEUDO OPERATIONS are discussed in detail in chapter 3.

2.1.3 MACHINE INSTRUCTIONS (OPERATIONS)

MICRO contains a predefined list of symbolic names that correspond to most of the machine operation codes of the "NOVA" computer architecture. This list may be completely replaced by a dynamically generated list from one or more micro-machine instruction set definition (DEF) files. Whenever a statement begins with a symbolic name that exists in this machine operation code list that statement is classified as a MACHINE INSTRUCTION.

Each operation code is provided with a set of format data which is used to determine how the assembler should further handle the processing of the statement. The final product of the processing of a MACHINE INSTRUCTION is the generation of one or more object program words.

2.1.4 DATA (CONSTANTS)

When a statement does not meet the basic qualifications for any of the classifications described above, it will be considered to be a DATA declaration statement. A DATA statement may contain several different DATA declarations, separated by commas. Each DATA declaration will produce one word of object program data. The term "CONSTANTS" is included here since it is frequently used in other literature to refer to data generating statements.

2.2 LEXICAL ANALYSIS

Lexical analysis is the process by which an assembler locates symbolic entities on the source statement record. Each of these entities will be referred to below as an "element". In MICRO, the lexical analyser can recognize three types of elements: NAMES, OPERATORS, and NUMBERS. The standard ASCII character set is used by MICRO, though only a 64 character subset (see APPENDIX A) is recognized during lexical analysis. The following subsections define these elements.

2.2.1 NAMES

A NAME is defined as a letter or period (.) character alone, or followed by any number of letters, digits, or periods. Although a NAME may be any length, only the first 10 characters of each NAME are used to uniquely identify that NAME. It is the programmer's responsibility to avoid using NAMES where only characters beyond the tenth position differ. Each NAME will take on further semantic meaning, depending on how it is used and what it represents. This will be discussed under STATEMENT PROCESSING, in section 2.3 below.

Examples: Legal NAMES:

A ABCDE00000 ANTIENERGISTIC ...AND.SO.FORTH...

Illegal NAMES:

987 <actually a number> F(X) <contains illegal characters>

777SEVEN <a name cannot begin with a digit>

2.2.2 OPERATORS

An OPERATOR is normally a single, special character (not a letter, digit, or period.) OPERATORS are used to delimit sets of elements (these sets are known as fields), or to define relationships between elements (such as arithmetic operations.) The following is a list of OPERATORS and their meanings to MICRO.

" "	Any number of blank characters may delimit specific fields.
!	Exclamation point, represents the logical operation "OR".
&	Ampersand, represents the logical operation "AND".
+	Plus, used for arithmetic addition.
-	Minus, used for arithmetic subtraction.
*	Asterisk, used for arithmetic multiplication.
/	Slash, used for arithmetic division.
,	Comma, used to delimit fields within certain statements.
;	Semicolon, used to delimit the comment field of a statement.
@	At sign, recognized during "NOVA" type assemblies (indirect).
#	Pound sign, recognized during "NOVA" assemblies (suppress data).
=	Equal, classifies symbolic EQUATE statements.
:	Colon, identifies symbolic LABEL names.
.	Period, special use to identify decimal valued NUMBERS.
"	Quotation mark, precedes character value constants.
<	Less than character, and...
>	Greater than character, delimit subfields in text strings.

Any other special characters not listed above will be lexically marked as OPERATORS, but will be indicated as erroneous elsewhere in MICRO.

2.2.3 NUMBERS

A NUMBER consists of a string of one or more digits, 0 through 9. NUMBERS may be subclassified as DECIMAL (base 10) NUMBERS whenever the digit string is terminated by a period (.). When delimited by any other character, the numeric value representation of a NUMBER is determined by the currently assigned radix (base) value. At the beginning of an assembly the initial radix value is 8, permitting all NUMBERS not followed by a period to be interpretted as octal values. NUMBERS are syntactically defined to legally consist of only those digits that are lower in value than the currently defined radix value.

Examples: Legal NUMBERS:

(Radix = 8)	32767.	32767	0010.	101010	2
(octal values)	077777	032767	000012	101010	000002
(Radix = 2)	32767.	32767	0010.	101010	2
(octal values)	077777	(illegal)	000012	000052	(illegal)

A numeric value representation of individual ASCII characters may be obtained by preceding each character by a quotation mark (") character. Thus the character pairs "A through "Z will produce the octal values 101 through 132, respectively. Refer to APPENDIX A for a complete list of ASCII character values.

2.3 EXPRESSION EVALUATION

An EXPRESSION is defined to be a set of elements consisting of NAMES and NUMBERS separated by any of the recognizable arithmetic and logical OPERATORS. The purpose of an EXPRESSION is to provide a means of producing a single numeric value through reference to, and arithmetic operations upon, other values.

When an EXPRESSION is processed, the elements are accessed in a left to right order. There are no hierarchal attributes assigned to any OPERATORS. NUMBERS are converted to binary values according to the rules covered in section 2.2.3. NAMES are located in the symbol table and their values are extracted. When a NAME is referenced it must be found in the symbol table, otherwise a diagnostic indication will be produced.

Most EXPRESSIONS are processed only during assembly PASS 1. These EXPRESSIONS may contain references to NAMES declared anywhere within the source program. Certain PSEUDO OPERATION statements are capable of processing EXPRESSIONS during PASS 1. NAMES referenced by these statements must be specified in the source program preceding the reference. Specification of these NAMES may be accomplished through use of LABEL and EQUATE statements, discussed in sections 2.4.1 and 2.4.2, respectively. The PSEUDO OPERATIONS that process EXPRESSIONS in PASS 1 are: .BLK first operand only (section 3.4.1), .IFE (section 3.2.1), .IFN (section 3.2.2), .LOC (section 3.4.2), .RDX (section 3.4.3).

2.3.1 EXPRESSION IN CONTEXT

There are two types of context to be considered. First is the assembly phase. During PASS 1, EXPRESSIONS will be processed only if they appear on certain CONTROL statements, in appropriate fields. All EXPRESSIONS are evaluated during PASS 2, even those already processed in PASS 1. The results of those evaluated in both passes must match or an error condition may be indicated. The second context is the location of the EXPRESSION, in terms of the particular statement class and field position on that statement.

An EXPRESSION field may contain any number of elements. It will be terminated upon occurrance of the delimiter OPERATORS: comma (,), semicolon (;), and greater than (>), and also by end of record. The use of particular delimiters depends on the type of statement used. For example, the greater than delimiter is used only within text strings, where an EXPRESSION can be imbedded. Blanks may be used as NAME or NUMBER delimiters, but cannot terminate an EXPRESSION.

2.3.2 SUPPORTED OPERATIONS

A total of six arithmetic and logical operations are supported by MICRO. The four arithmetic operations are: addition, subtraction, multiplication, and division. Two logical operations are provided: logical sum (OR) and logical product (AND). All operations are performed in a strict left to right order. The following list shows the OPERATORS used to represent these operations. Each is followed by an example (octal radix is assumed.)

\+ ADDITION
```
A + B + MORE + CONSTANTS + 1 + 2000.
```

\- SUBTRACTION
```
0 - A + B - 100. + X - Y          ; THE LEADING 0 MAY BE OMITTED.
```

* MULTIPLICATION
```
5 * DATA * OFFSET + 1
```

/ DIVISION
```
WORDS / 2 + 1
```

! LOGICAL SUM (OR)
```
A * B / C + 7 ! 400000            ; FORCE SIGN OF VALUE NEGATIVE.
```

& LOGICAL PRODUCT (AND)
```
LOCATION.A - LOCATION.B & 400000; EXTRACT ONLY THE RESULTING SIGN BIT.
```

2.3.3 ARITHMETIC PRECISION

MICRO maintains a full 18 bit value for each NAME and NUMBER processed. NUMBERS that have values greater than that which can be represented by an 18 bit field will be truncated, modulo 262,144. During EXPRESSION evaluation all overflow from multiplication, addition, or subtraction operations will be discarded. No indication of this form of truncation will be made. Illegal division operations, such as dividing by zero, will usually produce a minus 1 result and no error indication.

2.4.1 LABELS

LABEL statements consist of one or more LABEL NAMES. A LABEL NAME is any legal NAME followed by a colon (:) character. Each of the LABELS appearing on the same record will be assigned the current numeric value of the micro-location-counter. Any other statement class may follow a LABEL statement within the same record. LABEL NAMES, though, may not be preceded by any other statement class on a record.

LABELS declared ahead of statements that generate more than one word of object code will be assigned the location-counter value of the first word generated by that record. LABELS are used to provide symbolic reference to instructions or data areas.

Examples: LABEL statements:

```
START: BEGIN: ENTRY: ; ALL THREE LABELS WILL BE ASSIGNED THE SAME VALUE

K1000:        1000.  ; DATA CONSTANT VALUE 1000. IS NAMED K1000

K2000:        2000.  ; LABEL K2000 HAS A VALUE ONE GREATER THAN K1000
```

2.4.2 SYMBOLIC EQUATES

A statement beginning with a NAME followed by an equal sign (=) is classified as an EQUATE statement. These statements provide the capability to assign values to LABELS other than the value of the current micro-location-counter. A LABEL NAME may be assigned a value only once within the same assembly. The value to be assigned is specified in the field following the equal sign. This field contains one EXPRESSION. EXPRESSIONS are described in section 2.3, above.

Examples: EQUATE statements:

```
ABC  =    770.              ; LABEL "ABC" REPRESENTS 770 DECIMAL

CBS  =    ABC + 990. / 2.   ; LABEL "CBS" REPRESENTS 880 DECIMAL

NBC  =    CBS - ABC * 6.    ; LABEL "NBC" REPRESENTS 660 DECIMAL
```

2.4.3 PSEUDO OPERATIONS

When a statement begins with a NAME, and that NAME is in the PSEUDO OPERATIONS list, MICRO will perform some immediate, special, action. Most of these actions function to control MICRO's operation. Following recognition of the OPERATOR a CONTROL statement processor is entered to perform the specified action. LABELS may precede all statements in this class. An OPERATOR NAME must be followed by one or more blanks, a semicolon, or an end of record.

Fields to the right of the PSEUDO OPERATOR NAME are handled differently by each CONTROL statement processor. Some require full arithmetic EXPRESSIONS to define the value to be used. Others utilize the field as a character string, disabling the normal lexical meaning of most characters. Comments may be included at the end of most CONTROL statement types, by preceeding them with a semicolon (;). CONTROL statements are terminated only by the end of record being reached.

Examples: CONTROL statements:

```
        .LOC    . + 256.                ; ADVANCE PROGRAM LOCATION 256 WORDS

        .TXT    / CHARACTER STRING /;   GENERATE ASCII TEXT DATA
```

2.4.4 INSTRUCTION PROCESSING

When a statement begins with a NAME, and that NAME is in the OPERATION CODES list, MICRO will begin to generate a MACHINE INSTRUCTION. The action to be taken in the generation of each instruction depends on which of the 16 microinstruction formats, or 8 NOVA instruction types, has been declared. MICRO locates the instruction format associated with the OPERATOR NAME, within the OPERATION CODES list. The actual instruction format is passed to MICRO, from the DEF file, as a binary value. This value may be declared symbolically during a Nanoassembly. The table below indicates the binary format values, the symbolic NAMES available during Nanoassemblies, and the actual format configuration.

The present capabilities for object code generation provide for either 18 or 36 bit instructions, with several different operand field selections. Codes 40 through 47 generate 36 bit (2 word) instructions, while the remaining codes generate 18 bit (1 word) instructions. MICRO may be extended to support an unlimited number of instruction configurations, as the need arises.

OCTAL CODE	NANOASSEMBLY DEFINITION NAME	SOURCE STATEMENT FORMAT
20	OP NULL	OP7
21	OP A.B	OP7 A5A,B6A
22	OP A.BR	OP7 A5A,B6R
23	OP AR.B	OP7 A5P,B6A
24	OP AR.BR	OP7 A5P,B6R
25	OP ABR	OP7 AB11R
26	OP ABS	OP7 AB11A
27	OP A	OP7 A5A
30	OP AR	OP7 A5P
31	OP B	OP7 B6A
32	OP BR	OP7 B6R
33	-----	
34	-----	
35	-----	
36	-----	
37	-----	
40	OP A.B.V	OP7 A5A,B6A,V18A
41	OP ABR.V	OP7 AB11R,V18A
42	OP ABS.V	OP7 AB11A,V18A
43	OP ABCDE	OP7 A5A,B6A,C6A,A6A,B6A
44	OP ABC.DE	OP7 A5A,B6A,C6A,AB11R
45	-----	
46	-----	
47	-----	

KEY	Bit Positions
OP7	XXX XXX X00 000 000 000
A5	000 000 0XX XXX 000 000
B6	000 000 000 000 XXX XXX
AB11	000 000 0XX XXX XXX XXX
C6	XXX XXX 000 000 000 000
A6	000 000 XXX XXX 000 000
V18	XXX XXX XXX XXX XXX XXX

A = Absolute Arithmetic Value
R = Relative to Location Counter
P = Positive Location Counter Rel.

LABELS may precede all MACHINE INSTRUCTION statements. The OPERATION CODE NAME must be followed by one or more blanks, a semicolon, or an end of record.

2.4.5 DATA (CONSTANT) PROCESSING

Once a DATA statement is recognized each field on the statement will generate one word of data. Fields are separated by commas (,). EXPRESSION evaluation is performed on each field, and the resulting value is output in 18 bit format. LABELS may precede the first data field on the record. Comments may follow the last data field, when preceeded by a semicolon (;).

```
Examples:       DATA statements:

                1420                    ; SINGLE DATA WORD

   CONSTANTS:   1, 2, 3, 4              ; FOUR DATA WORDS, LABEL REFERS TO THE 1

   VALUE.2:     CONSTANTS + 1           ; CONTAINS LOCATION OF CONSTANT VALUE 2
```

MICRO does not support nesting of conditional tests. Any .IFE or .IFN statements encountered while in non-assembly state will be ignored. It is permissible to place several tests preceding one .ENDC statement, any one of which may enter non-assembly state.

3.2.1 .IFE - ASSEMBLE IF EXPRESSION IS EQUAL TO ZERO

```
.IFE        <EXPRESSION>          ; (COMMENTS)
```

If the result of the EXPRESSION is zero all source statements following this statement will continue to be assembled. If the result is any non-zero value then source statements will be skipped (and listed if listing is enabled) until an end of conditional assembly statement is encountered. An omitted OPERAND field will be recognized as an error.

Examples:

```
.IFE        1                     ; FORCED SKIP CONDITION.

.IFE        SECOND - FIRST - 2    ; ASSEMBLE IF SECOND = FIRST + 2.

.IFE        EVALUED & 400000      ; ASSEMBLE IF "EVALUED" IS POSITIVE.
```

3.2.2 .IFN - ASSEMBLE IF EXPRESSION IS NON-ZERO

```
.IFN        <EXPRESSION>          ; (COMMENTS)
```

.IFN is the inverse function to .IFE, above. Source statements following an .IFN statement will continue to be assembled if the result of the EXPRESSION is non-zero. When the result is zero then source statements will be skipped, and listed if listing is enabled, until an end of conditional assembly statement is encountered. An omitted OPERAND field will be recognized as an error.

Examples:

```
.IFN        0                     ; FORCED SKIP CONDITION.

.IFN        OPTIONS & 5           ; ASSEMBLE IF BITS 0 OR 2 ARE PRESENT.

.IFN        EVALUED & 400000      ; ASSEMBLE IF "EVALUED" IS NEGATIVE.
```

3.2.3 .ENDC - END CONDITIONAL ASSEMBLY

```
.ENDC                                   ; (COMMENTS)
```

The .ENDC statement simply reinstates normal source statement assembly, whenever the non-assembly state is in effect. Otherwise, it has no effect.

Example:

```
.IFN        A-B & 400000        ; IF "A" < "B", GENERATE AN ERROR
(ERROR "A" LESS THAN "B")       ;  MESSAGE FROM "MICRO" VIA UNKNOWN
.ENDC                           ;   STATEMENT TYPE.
```

3.3 LISTING CONTROL

Three PSEUDO OPERATIONS are provided for control of the source program listing: EJECT, LISTOFF, and LISTON. These statements will ignore their OPERAND fields. Listing selection may be controlled conditionally, using the CONDITIONAL PSEUDO OPERATIONS described in section 3.2. The EJECT statement will be recognized unconditionally, in order to permit uniform listing appearance while in non-assembly mode. When the full listing is suppressed, and only lines in error are to be displayed, the function of EJECT is also suppressed. Erroneous lines will be displayed no matter what listing selections have been made. Only a SUPPRESS ALL LISTABLE OUTPUT option, specified during assembly invocation (see section 7.3), will suppress lines in error.

3.3.1 EJECT - END OF PAGE (FORMS)

```
EJECT                                   ; (COMMENTS)
```

The EJECT statement will set an indicator, in MICRO, to list the next line of listable output on a new page. This statement will have no effect when the listing is being displayed on the alternate list device, normally the system console, as specified by a SELECT ALTERNATE DEVICE option during assembly invocation. It will also be ignored when it follows a LISTOFF statement, described in section 3.3.2 below.

Example:

```
EJECT                                   ; END OF THIS PAGE.................
```

3.3.2 LISTOFF - TURN LISTING OFF

```
   LISTOFF                          ; (COMMENTS)
```

The LISTOFF statement will suppress listing of all following statements, until a LISTON statement (section 3.3.3) is encountered. Any EJECT statements located between a LISTOFF - LISTON statement set will be suppressed.

Example:

```
   LISTOFF                          ; NO LISTING FOR A WHILE.
  .........
```

3.3.3 LISTON - TURN LISTING ON

```
   LISTON                           ; (COMMENTS)
```

The LISTON statement restores listing of all following source statements, and recognition of EJECT statements. LISTON will always be listed. It will have no effect if encountered when listing is already enabled.

Example:

```
   .IFE        LIST-1               ; ASSEMBLE NEXT STATEMENT IF LIST = 1.
   LISTOFF                          ;  LISTING SUPPRESS OPTION IS SELECTED.
   .ENDC                            ;   END CONDITIONAL SECTION.
  /............\
 /..............\  THESE STATEMENTS, AS WELL AS THE .ENDC
 \................/  ABOVE, ARE CONDITIONALLY LISTED.
  \............/
   LISTON                           ; RESTORE LISTING AT THIS POINT.
```

3.4 ASSEMBLY CONTROL

The following six PSEUDO OPERATIONS provide general control over the operation of MICRO. The three statements: .BLK, .TXTM, and .TXT, provide for the generation of multiple word data arrays. The remaining three statements: .LOC, .RDX, and .XPNG, provide control over source program interpretation and symbol table content.

3.4.1 .BLK - BLOCK DATA GENERATION

```
   .BLK        <EXPRESSION>(,<EXPRESSION>) ; (COMMENTS)
```

The first OPERAND field of the .BLK statement is required, and specifies the number of words to be occupied by this data array. If the, optional, second field is specified then the resulting 18 bit value will be used to fill all locations of the data array. Otherwise, no data will be generated and only the micro-location-counter will be adjusted to reflect the space required. The first OPERAND field is evaluated during PASS 1, and therefore may only reference LABELS defined ahead of the .BLK statement. The second OPERAND field is evaluated during PASS 2, and may reference any properly defined LABELS.

Examples:

```
CARDS:  .BLK    80./2                   ; ALLOCATE 80 CHARACTER DATA BUFFER.

TABLE:  .BLK    TABSIZE,-1              ; GENERATE TABLE, INITIALIZED TO -1'S.

SYMBOL: .BLK    1000.*8.                ; DEFINE SYMBOL TABLE ARRAY, 1000 X 8.
```

3.4.2 .LOC - LOCATION COUNTER DEFINITION

```
   .LOC         <EXPRESSION>            ; (COMMENTS)
```

Normally the micro-location-counter is updated following the processing of each statement capable of generating object code or data. Standard MACHINE INSTRUCTIONS will occupy one or two words of storage, and will increment the micro-location-counter appropriately. Other PSEUDO OPERATIONS will produce variable amounts of object data, also incrementing the location-counter. When it is desired to place the location-counter at some address that is not a representation of generated object storage the .LOC PSEUDO OPERATION may be used. The result of the EXPRESSION evaluation will be placed directly into the micro-location-counter. The EXPRESSION is processed during assembly PASS 1, and may reference only those LABELS defined preceding the .LOC statement.

.LOC may be used at the beginning of the program to set the initial start of program address. Later on it may be used to allocate space, similar to the .BLK statement (section 3.4.1), through use of an EXPRESSION adding a constant to the current micro-location-counter. The current micro-location-counter is accessible at any time by reference to the special LABEL ".".

Examples:

```
        .LOC    256.                    ; BEGIN PROGRAM AT LOCATION 256.

CARDS:  .LOC    80./2.+.                ; DEFINE 80 CHARACTER DATA BUFFER.

PRESENT:.LOC    .-SOMETHING             ; IDENTIFY "PRESENT" LOCATION, THEN GO
BACK:   SOME.DATA                       ;  BACK TO INSERT SOME DATA, AND FINALLY
        .LOC    PRESENT                 ;   COME BACK TO THE "PRESENT".
```

3.4.3 .RDX - NUMBER BASE DEFINITION

```
   .RDX         <DECIMAL EXPRESSION> ; (COMMENTS)
```

NUMBERS are initially decoded as octal value representations, unless the NUMBER is terminated by a period (.) which represents decimal value. The numeric base, or radix, which MICRO uses to decode NUMBERS is itself variable, and may be changed at any time through the .RDX PSEUDO OPERATION. The OPERAND field is represented by a DECIMAL EXPRESSION, which means that within this field all NUMBERS will be decoded as having decimal values. This EXPRESSION is processed during PASS 1, and should reference only predefined LABELS. Any radix value between 0 and 10 will be considered legal by MICRO.

Examples:

```
        .RDX    10                      ; ALL NUMBERS WILL BE DECIMAL.

        .RDX    2                       ; ALLOW GENERATION OF BINARY BIT VALUES.

        .RDX    8                       ; RESTORE OCTAL NOTATION.
```

3.4.4 .TXTM - SET TEXT MODE

```
    .TXTM       <EXPRESSION>            ; (COMMENTS)
```

Text, or character string, data may be generated by any of the text PSEUDO OPERATIONS discussed in section 3.4.5, below. All text generated by MICRO is produced in 7-bit ASCII, as shown in APPENDIX A, CHARACTER SET. This text data will be packed, two characters to a word, using the rightmost 16 bits as two 8 bit character fields. The default ordering of characters within a word is right to left. The first character is placed into bit positions 7 to 0, and the second is placed into positions 15 to 8. When it is desired to generate characters within each word in a left to right order .TXTM must be used. Any non-zero EXPRESSION result will indicate left to right ordering, while zero will produce the default right to left order.

Examples:

```
        .TXTM   1                       ; TEXT WILL BE LEFT TO RIGHT.

        .TXTM   0                       ; TEXT WILL BE RIGHT TO LEFT.
```

3.4.5 .TXT - GENERATE TEXT STRING DATA

```
    .TXT        $<STRING!<EXPRESSION>>$ ; (COMMENTS)--ZERO PARITY
    .TXTO           "           "       ;       "    --ODD  PARITY
    .TXTE           "           "       ;       "    --EVEN PARITY
    .TXTF           "           "       ;       "    --ONE  PARITY
```

Character string, text, data may be generated through use of any of the .TXT PSEUDO OPERATIONS shown above. Text is generated in 8 bit character fields, placed two to a word, within bit positions 15 through 0. MICRO will produce characters in 7 bit ASCII format; see APPENDIX A, CHARACTER SET. Character string object data will always be terminated by at least one 8 bit zero character. If a string contains an odd number of characters then the last character of the last word will be zero. If an even number of characters are defined then MICRO will, automatically, generate an extra word of zeroes after the string.

The first non-blank character following the PSEUDO OPERATION NAME will be recognized as the string delimiter, as represented by the dollar sign ($) above. The second encounter of this character will terminate the character string. The delimiters will not be included in the generated string data. A comment field may follow this terminator. When it is desired to include characters that are not in the ASCII character subset, or to include the string delimiter character itself, a numeric EXPRESSION may be imbedded within the string to accomplish this. Each EXPRESSION field is preceded by

a less-than character (<), and is terminated by a greater-than character (>). The result of the EXPRESSION will be truncated to 8 bits, and will be placed within one object character field. A character constant, a quotation mark (") preceding any character, may then be used to represent the delimiter character. Symbolic representations may then be used for the special codes needed, such as carriage returns or communications control codes.

Since the ASCII character data is representable in only 7 bits, the eigth bit is frequently used for parity error checking. The parity bit will be maintained as zero when generating text with the basic .TXT statement. If odd, even, or one parity bit encodings are desired then one of the alternate text operations should be used as indicated above.

Examples:

```
MESSAGE:.TXT    /LAST LINE RECEIVED WAS IN ERROR!/

*       GENERATE 80 CHARACTERS OF ASCENDING NUMERIC ORDER.
ALIGN:  .TXT    !0123456789 123456789 123456789 123456789! ; FIRST 40...
        .LOC    .-1                     ; BACK UP OVER THE GENERATED ZERO WORD.
        .TXT    ! 123456789 123456789 123456789 123456789! ; SECOND 40...

CARRIAGE.RETURN =       15              ; RETURN TO BEGINNING OF LINE.
LINE.FEED =             12              ; SPACE ONE LINE VERTICAL.
        .TXTM   1                       ; LEFT TO RIGHT FOR CONSOLE MESSAGE.
HELP:   .TXTO   '<CARRIAGE.RETURN><LINE.FEED>*** PLEASE CHANGE PAPER'

DEMAND: .TXT    /<15><12>RESPOND AFTER THE "<"/>" CHARACTER <15><12><"/>/
*  PRODUCES THE FOLLOWING OBJECT STRING (OCTAL):
*       015 012, 122 105, 123 120, 117 116, 104 040, 101 106, 124 105,
*       122 040, 124 110, 105 040, 047 057, 047 040, 103 110, 101 122,
*       101 103, 124 105, 122 040, 015 012, 057 000.
```

3.4.6 .XPNG - DELETE PREDEFINED SYMBOLS

```
   .XPNG                                ; (COMMENTS)
```

During assembly PASS 1, the .XPNG statement will delete all LABEL NAMES and MACHINE INSTRUCTION definitions from the assembler symbol table. PSEUDO OPERATION NAMES are not disturbed. .XPNG is provided for compatibility with NOVA type assemblers. Following its use there are no instruction OPERATION CODE NAMES available, and only DATA statements can be processed.

This statement is meant to be used only at the beginning of an assembly, and would be followed by symbolic MACHINE INSTRUCTION operation code, and format, definitions. Symbolic definition is not available in this version of the MICRO Assembler.

4 INSTRUCTION SETS

Any number of instruction sets may be processed by MICRO. When invoked, the assembler contains a predefined instruction set for the NOVA type system architecture. This is provided in order to support NCS, the system which supports QM/SYSTEM assemblers and data file management. Other micro-instruction sets are defined through DEFINITION files, prepared during Nanoassemblies. Actual, detailed, discussions of DEFINITION file and microinstruction set development are available in the QM NANOASSEMBLER REFERENCE MANUAL.

This chapter briefly discusses the rules for utilizing the various instruction sets. The access to DEFINITION files is also covered.

4.1 PREDEFINED "NOVA" INSTRUCTION SET

In MICRO, the NOVA instruction set consists of 182 MACHINE INSTRUCTION definitions. Most of these are identical to the NOVA instruction set provided by standard NOVA assemblers. User's should be fully familiar with the NOVA architecture before using this instruction set.

Several special definitions are provided for the support of NCS, and for ease of utilization of the arithmetic comparison instructions. Only these special definitions are listed below.

4.1.1 MACHINE INSTRUCTIONS

NOVA operations are provided for the NOVA 1200, and NOVA 800 machine series. The QM/SYSTEM NOVA emulation provides all capabilities, including support for the Hardware Multiply/Divide option. Input/output devices are driven according to actual hardware device specifications. See the NCS OPERATIONS GUIDE for further details. The table below lists those predefined OPERATION CODE NAMES which are provided for NCS support.

Conditional and unconditional skip/no-skip instructions:

NOP		NO-OPERATION (ACTUALLY A MOV# 0,0)
SEQ	A,B	SKIP NEXT INSTRUCTION IF A .EQ. B
.EQ.	A,B	" " " " "
SGE	A,B	SKIP NEXT INSTRUCTION IF A .GE. B
.GE.	A,B	" " " " "
SGT	A,B	SKIP NEXT INSTRUCTION IF A .GT. B
.GT.	A,B	" " " " "
SLE	A,B	SKIP NEXT INSTRUCTION IF A .LE. B
.LE.	A,B	" " " " "

```
SLT     A,B             SKIP NEXT INSTRUCTION IF A .LT. B
.LT.    A,B               "    "       "       "     "

SNE     A,B             SKIP NEXT INSTRUCTION IF A .NE. B
.NE.    A,B               "    "       "       "     "
```

Abbreviations used in the above list are:

.EQ.	EQUAL TO
.GE.	GREATER THAN OR EQUAL TO
.GT.	GREATER THAN
.LE.	LESS THAN OR EQUAL TO
.LT.	LESS THAN
.NE.	NOT EQUAL TO

4.1.2 STATEMENT FORMATS

All NOVA instructions are one word in length. There can be from zero to three OPERAND fields per instruction. In several instruction formats, one or more, fields may be optional. The default value for optional fields is zero. Use of the special OPERATORS for indirect addressing (at-sign, @), and suppress data transfer (pound-sign, #) are restricted as follows.

The indirect address form of DATA statement requires that the at-sign be placed at the beginning of the first element of the DATA DEFINITION field. This will result in presetting an initial OPERAND value of 100000 (octal), which represents setting the indirect address bit of the NOVA data word. This method of address generation restricts the remainder of the EXPRESSION to consist only of addition and subtraction operations, none of which may cause field overflow. If multiplication or division is required in an indirect address definition the at-sign should not be used, and a constant value of 100000 should be logically added at the end of the EXPRESSION.

Examples:

```
        @DATARRAY + I                   ; NORMAL APPEARANCE OF INDIRECT ADDRESS.

        DATUM * 4 + BASE ! 100000       ; ALTERNATE INDIRECT ADDRESS FORMAT.
```

Indirect address reference from an instruction OPERAND field also requires that the at-sign be placed just ahead of the first element of the address field. Placement of the indirect OPERATOR at some other point will cause the address EXPRESSION to evaluate incorrectly.

Examples:

```
        LDA     0,@STACK                ; LOAD DATA FROM ADDRESS AT "STACK".

        STA     2,@4,3                  ; STORE DATA AT THE ADDRESS FOUND AT
                                        ;  THE LOCATION DEFINED BY THE CONTENTS
                                        ;   OF ACCUMULATOR(3) + 4.

        JMP     @RETURN                 ; TRANSFER TO THE ADDRESS IN MEMORY
                                        ;  LOCATION "RETURN".
```

The suppress data transfer OPERATOR may be appended to any of the logical or arithmetic, register to register, instruction OPERATION CODE NAMES. The pound-sign character will indicate an error if used elsewhere on an instruction statement.

Examples:

```
MOV#   1,1,SZR          ; SKIP IF ACCUMULATOR 1 IS ZERO.

SUBO#  1,2,SNZ          ; SKIP IF AC 1 IS NOT EQUAL TO AC 2.
```

4.2 EXTERNALLY DEFINED MICROINSTRUCTION SETS

A single instruction DEFINITION file (DEF) usually contains the complete structural definition of a microinstruction set. When two, or more, microinstruction sets are to be used concurrently all must be declared as DEF files during assembly invocation. Microinstructions are implemented as nanoprograms within the DEF file. Each nanoprogram will be loaded at an absolute location in Nanostore, and therefore has an absolute address value which becomes the OPERATION CODE value. When multiple microinstruction sets are used they must be allocated to unique addresses within Nanostore. Specification of DEF files is covered in chapter 7, OPERATING PROCEDURE.

4.2.1 MACHINE INSTRUCTIONS

As an example, the "MULTI" Micromachine is used as a common microprogramming base for many emulations, user processes, and for the "TASK" microprogrammed OPERATING SYSTEM (see section 5.2). "MULTI" is fully described in the "MULTI MICROMACHINE DESCRIPTION" Manual. It consists of 78 microinstructions, which are similar in appearance to conventional machine instruction sets. Combining "MULTI" with a specialized microinstruction set for a particular process will produce an efficiently operating program, which may be quickly implemented.

4.2.2 STATEMENT FORMATS

The formats for microinstruction generation are discussed in section 2.4.4, above. All OPERAND fields defined for a specific instruction must be specified on the MACHINE INSTRUCTION statement. Null fields are permitted, and will produce zero results. There are two classes of fields processed by MICRO: absolute value, and location-counter relative. The contents of an absolute value field may be positive or negative, as long as the two's complement positive value will fit within the field width. Location-counter relative fields may be either one way, forward or reverse reference, or two way, where positive means forward reference and negative means reverse. One way relative fields always contain absolute values, while the signed relative fields may contain two's complement values. Values placed into location-counter relative fields must reference program addresses where the displacement value is small enough to fit within the field width provided.

Field width is defined by the specific field attribute identification, which also specifies field location within the instruction word being defined. Only those field structures are supported that correspond to the QM/SYSTEM microprogramming architecture, as defined in the "HARDWARE LEVEL USER'S MANUAL". These field formats are also depicted in section 2.4.4, above. In this version of MICRO the micro-location-counter value always points to the word being generated. Therefore, when the location-counter is referenced within a 36 bit wide instruction (through LABEL NAME "."), it will point to the first word of the instruction only when used in fields defining that word, and will contain the second word address when referenced in fields defining the second word.

4.3 DEFINITION FILES

Each DEFINITION file is made up of both the actual object Nanoprogram code and symbolic information for use by the Microassembler. Special files can be generated that contain only the symbolic information, which will reduce the amount of time MICRO spends in scanning the file while constructing its initial symbol tables. A symbolic definition is made up of ten 16 bit words; containing the symbol NAME (of up to 10 ASCII characters), the 18 bit object value, the microinstruction format characteristic, structural identification, and error flag fields.

Disk space allocations for DEFINITION files can be approximated by multiplying the number of symbols by 10 words, and then dividing by 256 words to determine the number of sectors needed. When the file also contains object Nanocode its size will grow substantially, requiring four 16 bit words per Nano-primitive field defined. There is no simple way of predicting file size in this case. Section 6.3 further describes the physical organization of the DEFINITION file.

5 PROGRAM STRUCTURE

This chapter discusses the organization of programs, and the interfacing of these programs to the various levels of Operating Systems provided with the QM/SYSTEM.

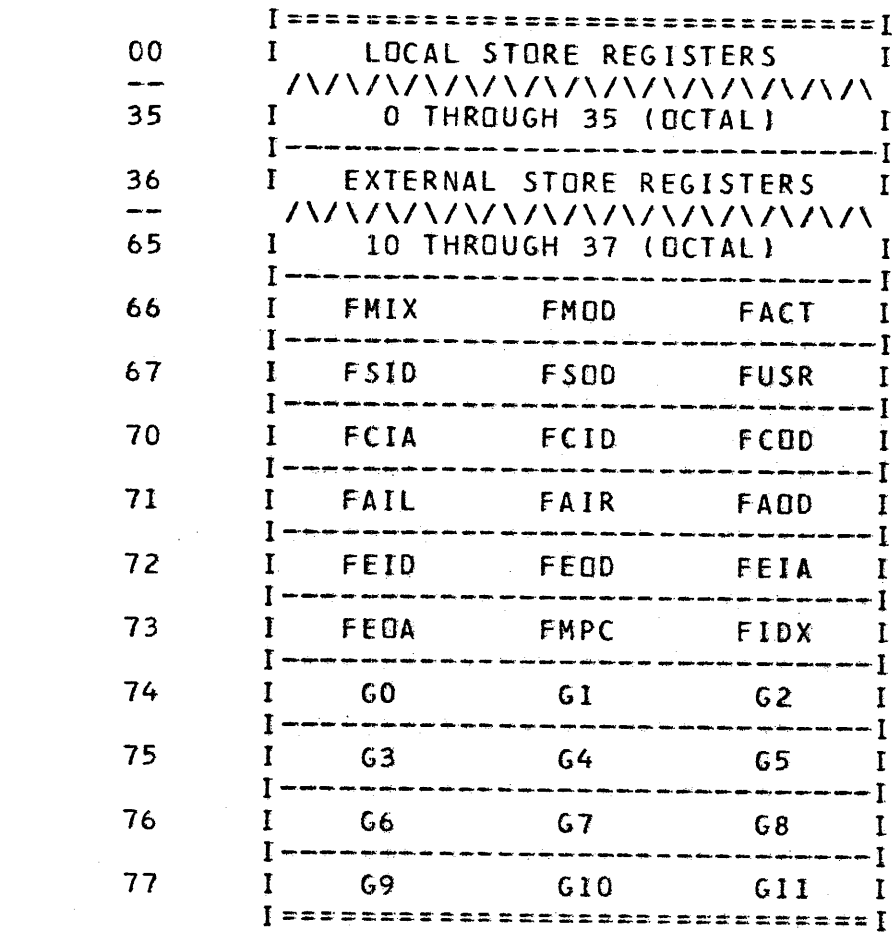
5.1 ABSOLUTE PROGRAMS

MICRO produces object programs in absolute format. This means that the data structures placed onto the BIN file will be loaded and executed in an unaltered fashion. Each source program must contain an initial location-counter declaration (.LOC), to define its operating storage load point. If no initial location is defined the program will be assembled to load at location 0.

5.1.1 MICROPROGRAMMING CONSTRUCTS

There are two methods of executing microprograms: as stand-alone, fully self-contained, microprograms, or as tasks operating under control of the "TASK" OPERATING SYSTEM. When run in stand-alone format, programs may be loaded directly from their BIN object files, through the "QMLD" microprogram loader, or from the N2022 Cartridge Tape System, as prepared using the "PREP" Cartridge Tape writing system (these are described in the "NCS OPERATIONS GUIDE".)

If the Nanoprogram System utilized by the microprogram being run permits initial program entry via direct transfer to a Nanostore location, then no special additional preparations are of concern to the microprogrammer. All QM/SYSTEMS contain a Read-Only-Memory, used in starting machine operation. This memory contains an instruction called MICROSTART, which may be called on to preset all QM/SYSTEM Local Store, External Store, and F Store registers to values placed within a 64 (decimal) word Control Store array. This MICROSTART array is organized as follows:



When starting any microprogram several of these registers must be defined. All undefined registers should be set to zero. The following information must be provided for a valid microprogram start:

FMPC Must indicate the Local Store register number to be used as the initial Micro-Program-Counter, 30, 31, 32, or 33 (octal).

FIDX Must contain the appropriate Nanostore page number, for proper microinstruction OPERATION CODE decoding (bits 2 - 0). FIDX also contains the ALU width, System Supervisor State, and Read-Only-Nanostore access controls. Refer to the "HARDWARE LEVEL USER'S MANUAL" for further details.

F*** Any other F Registers, as required by the Nanosystem conventions in use. (ie: FCOD and FAIR may need to be preset to 31.)

R(FMPC) Local Store Register specified in FMPC, as the initial Micro-Program-Counter. This register contains the address of the first microinstruction to be executed.

A MICROSTART Array may be placed anywhere within the microprogram. It should be placed on a 100 (octal) word boundary, to permit direct system program entry via QM/CPU F-Switch selection (see "NCS OPERATIONS GUIDE".) A version of the MICROSTART Array is utilized when starting a primary task under the "TASK" System.

5.1.2 "NOVA" PROGRAMMING CONSTRUCTS

NCS executes NOVA programs at the Main Store level of the QM/SYSTEM. The object program loader "LD" may be invoked by entering the command:

```
!LD,filename(-negdisplacement).
```

Where "filename" identifies the BIN file containing the object program. The optional "-negdisplacement" parameter is provided for NCS System Transient program support. It subtracts the octal value "negdisplacement" from all object program load addresses, placing the program lower in storage than the assembled load point.

NCS has no low storage requirements. Therefore, user programs may utilize locations 0 through the base of the NCS resident control program at 17077 (octal). The Resident occupies 17100 to 17777, making locations 20000 through the end of accessible storage available to user programs. Section 5.3, below, lists the NCS facilities provided for utilization of supported input/output drivers and program loaders.

5.2 "TASK" SYSTEM INTERFACE

For a microprogram to run under "TASK" it must first be placed into the active Task User Library. These libraries are usually resident in Main Store, in an area only accessible to the system. When in operation, microprograms may be broken up into SEGMENTS. These SEGMENTS may then be run sequentially as overlays to each other, or concurrently at different TASK Priority Levels. "TASK" operation is described in the "TASK CONTROL PROGRAM OVERVIEW" manual.

All facilities provided by TASK are accessed through the "SYSTEM" microinstruction, of the "MULTI" Micromachine, described below.

5.2.1 MEMORY ORGANIZATION

Under basic versions of "TASK" the Task Control Program, Control Store resident, occupies the highest region of available Control Store. All storage below this region is available to user programs, system processors, or combinations of user and system programs. The predefined NAME "TASK.BASE" passes the actual base address of the Resident to each microprogram assembly. This NAME is defined within the "MULTI" microinstruction set DEFINITION file.

Frequently, the active microprogram will utilize the console display and control functions of "PROD", the Programmable Run-time Operator Display program. PROD occupies the Control Store region immediately below TASK. When active, PROD executes as the highest priority Task. The primary purpose of PROD is control over the user microprogram, with a highly flexible debug and analysis capability. PROD is described in the "PROD USER'S GUIDE".

Main Store contains the TASK System and User SEGMENT overlay libraries. These libraries occupy the lowest region of physical Main Store. Every individual Task has access to its own region of storage, controlled by TASK and the Base / Field Length hardware feature of the QM/SYSTEM Central Processor. When necessary, user storage regions may overlap each other, or be fully contained within another region. This permits environments, such as an emulator, to have Tasks and Subtasks with hierarchal storage access privileges as well as the usual CPU access priorities. For example, the inner Task region may be the actual emulator, while the next outer Task may be an I/O device emulator which needs access to both its own working storage and the emulated storage space.

5.2.2 "SYSTEM" INSTRUCTION CALLS

To the "MULTI" microinstruction set the "SYSTEM" instruction is simply a programmed interrupt, received by the TASK Control Program. The OPERAND field is passed to TASK, which will interpret the parameter value as a specific function call. The following list describes several of the functions supported by TASK. A more complete explanation will be found in the "TASK CONTROL PROGRAM OVERVIEW" manual.

NAME	OCTAL CODE	FUNCTION
SYS.RCL+S	00	Terminates the current Task, passing the STOP CODE value "S" (0 to 15) to the Parent Task.
SYS.SIO	20	Start I/O operation. An I/O Control Block must be be identified through a Local Store register.
SYS.WAIT	40	Wait, and remove Task from access to the Central Processor, until the EVENT, identified by an Event Control Word, has occurred.
SYS.WAITL	41	Wait, and remove Task from access to the Central Processor, until one of the EVENTS, identified by a list of Event Control Word addresses pointed to by a Local Store register, has occurred.
SYS.TIME	47	Read the DATE and TIME registers, to Local Store.
SYS.ITASK	50	Initialize a Subtask's registers, and prepare it for execution.
SYS.STASK	51	Enable a Task for access to the Central Processor, as soon as it attains highest Task priority.
SYS.KTASK	52	Disable a Task from any further execution.
SYS.SMSB	55	Offset Main Store base address within allocation.
SYS.LOAD	60	Load an overlay SEGMENT, into Control Store, as identified by a Load Control Block pointed to through a Local Store register.

5.3 "NCS" SYSTEM INTERFACE

NCS is a basic operating environment originally developed for smaller memory computer installations. Version 1 of NCS was developed for use with a minimum NOVA architecture computer, using standard and specialized I/O peripheral equipment. The primary intent of NCS was to support data and program files in a small disk storage environment, with absolute minimal memory overhead. Version 1 requires a nominal 700 (octal) locations for its basic resident control programs, and 2000 (octal) locations in order to fetch a user program and initiate its execution. This represents a program initiation overhead of 12.5%, and a dynamic memory availability of better than 94%.

NCS is a transient-disk-task oriented system. Each system transient performs a specific task for the console operator, a user program, or for other transients. Transients are fetched from disk by a resident function called LDR. All transients are first loaded into an absolute location, known as the transient buffer, below the high core Resident Supervisor. Control information, found at the head of each transient, tells the loader whether the transient should be copied to another location in memory, and where, and whether execution control should be turned over to the transient, and again where. In the case of non-executable, multiple subroutine, transients a third piece of information, transient length, is provided to permit copying of only the active portion of the code within those relocatable transients. Transients may be no larger than one disk sector (400 octal locations). Transient format is shown in figure 5.3.A. Figure 5.3.AA represents the layout of an executable system task (Type 1 Transient). Figure 5.3.AB represents the structure of a multiple entry point subroutine group (Type 2 Transient).

The permanently resident segment of NCS occupies 700 (octal) locations of NOVA memory. Figure 5.3.B shows the layout of NCS Resident, using an 8K configuration for address examples. The left column of numbers indicates the base of each coded section. The rightmost column indicates the displacement of each section from the assembly language symbol "TOP" (which represents actual core size).

5.3.1 MEMORY ORGANIZATION

Locations TOP-1 through TOP-3 are jump instructions into primary NCS functions. Transfer is made to TOP-1 (EOJ) whenever a user program, or operator console task, wishes to terminate operation. EOJ calls on the supervisor control program, which places both CRT Console and TTY into command state. This location is also set into the NOVA console switches when first starting up the system (location 17777 in the 8K version).

Initiation of most system tasks is accomplished through transfer to TOP-2 (LDR), which unconditionally fetches a system transient sector. The calling sequence for LDR action is described in section 1.3. TOP-3 (LDRX) is used only during internal calls made by transients. LDRX calling sequence differs from the LDR sequence, in that part of the preceeding LDR call remains residually active during the LDRX activity. TOP-4 (RECUR) is normally used during an LDRX activity and provides the name of the originating task, which is to be recalled at completion of the current task.

During LDR operation accesses are made to disk directories, established in cylinder 0 of each disk. To permit some internal system tasks to use the existing LDR disk accessing routines for their own disk I/O, a pointer to the LDR disk command array is maintained at TOP-205. Only internal tasks should modify this array. General control over LDR actions is supported through a "MODE" control word, at location TOP-206. At normal job start "MODE" is automatically set to 0. The setting of "MODE" to non-zero will cause LDR to bypass any standard relocation of transients from the transient buffer area. It also suppresses execution of executable (Type 1) transients. It is the responsibility of the program in operation to reset "MODE" to 0. EOJ will also clear "MODE".

Operator console communication is maintained (optionally) through the "GETC" and "PUTC" routines within the resident. "PUTC" will transmit one ASCII character to the currently active console. Transfer to "PUTC" is through location TOP-207. "GETC" reads one character from any active console. It is entered by transfer to TOP-210.

Below NCS Resident at location 17100 (octal) are the standard disk interface routines. Entry is made at locations TOP-677 or TOP-700. Locations TOP-740 through TOP-701 are used for general program/system communication, and for inter-transient communication. The most frequent use for this area is for File Control Block (FCB) tables. An FCB is set up for each open disk file, during program execution. Each FCB identifies the current position, extent, buffer location, buffer content and file type for the active file.

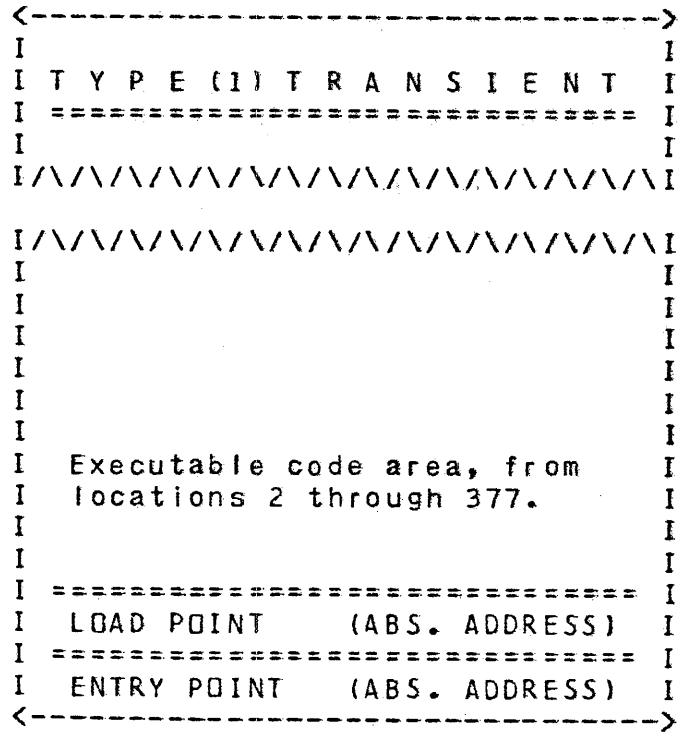


FIGURE 5.3.AA

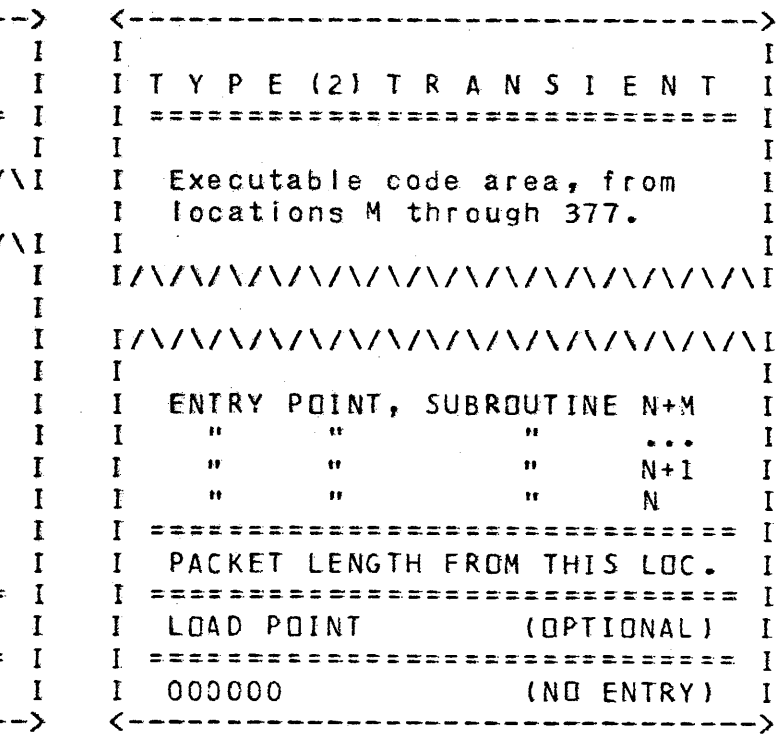


FIGURE 5.3.AB

FIGURE 5.3.A

EXAMPLE OF 8K MEMORY SYSTEM (NCS RESIDENT STRUCTURE)

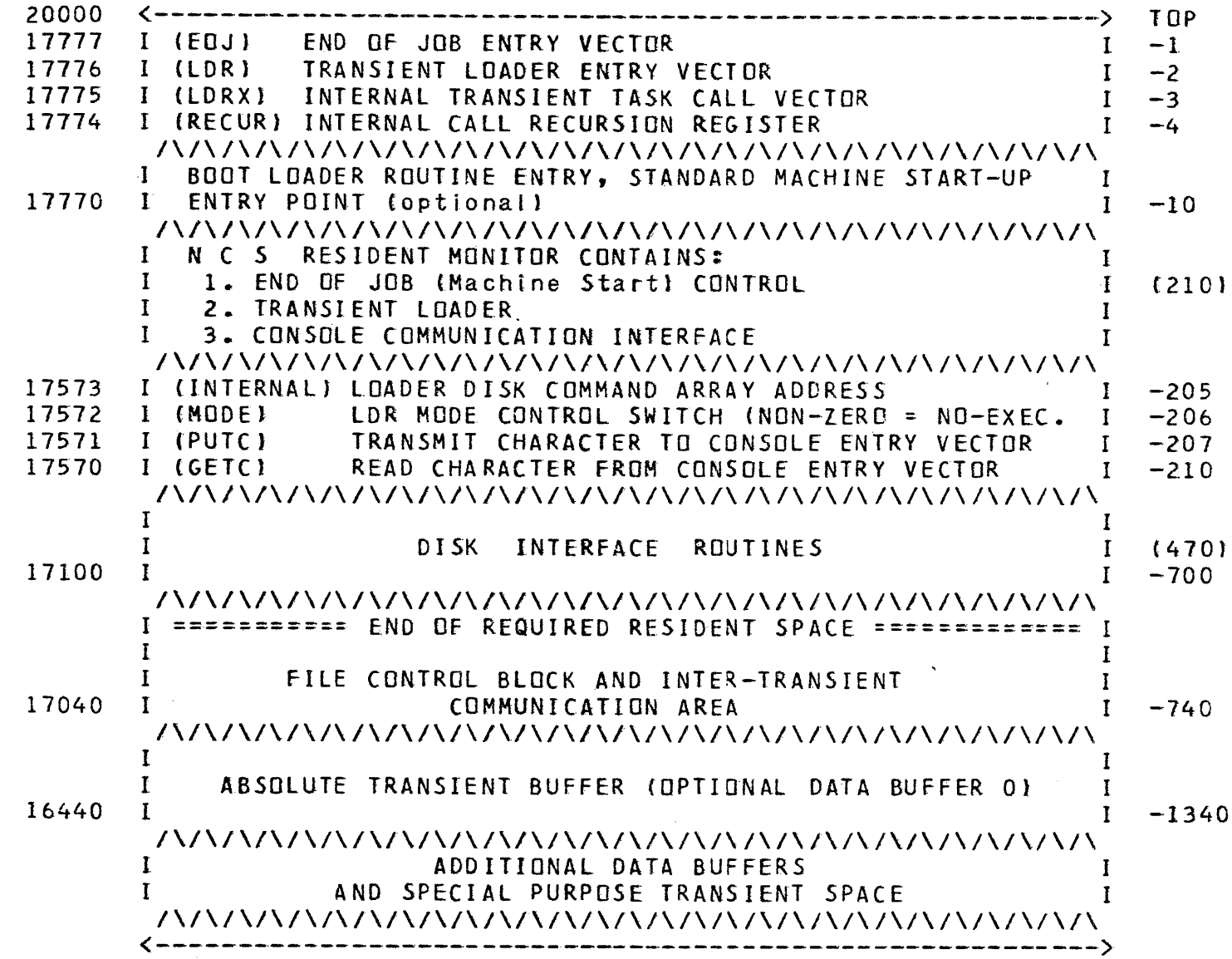


FIGURE 5.3.B

5.3.2 SYSTEM CALLS AND LINKAGES

This section describes the general transient and subroutine linkages found in NCS. Most linkages are shown below with minimal narrative. NOVA assembly language statements are used throughout the examples.

1. Normal transient call is through LDR. Following transient fetch, execution will depend on the type of transient called, and whether "MODE" is set to enable execution. Whenever execution is inhibited, transfer returns to CALL + 3. Return to CALL + 2 occurs only when there has been a physical disk error returned by the disk driver routines. The ASCII code name for the transient to be executed is always passed as location CALL + 1.

```
CALL:   JSR     @LDR        ; CALL LOADER
        FUNCT               ;  TRANSIENT NAME
        JMP     ERR         ;   DISK ACCESS ERROR RETURN
        JMP     NORMAL      ;    NORMAL RETURN
LDR:    TOP-2               ;     LDR ENTRY POINT ADDRESS
```

2. Normal return may occur following completion of the execution of a type 1 transient. Each transient may return to alternate locations to indicate the results of its activity:

```
        JSR     @LDR        ; CALL LOADER TO EXECUTE
        ":                  ;  OPEN FILE FUNCTION (:)
        JMP     ERROR       ;   DISK ERROR OCCURRED ON EITHER
                            ;    TRANSIENT LOAD OR DIRECTORY SEARCH.
        JMP     NONE        ;     RETURN HERE IF NOT ON FILE.
        JMP     FOUND       ;      NORMAL RETURN AFTER FILE OPEN.
```

3. Internal transient to transient call:

```
        LDA     0,FUNCT     ; SET TRANSIENT NAME IN AC 0.
        JMP     @LDRX       ;  CALL INTERNAL LOADER. (THERE IS NO
                            ;   RETURN FROM THIS CALLING SEQUENCE).
LDRX:   TOP-3               ;    LDRX ENTRY POINT VECTOR ADDRESS.
FUNCT:  "Q                  ;     TRANSIENT NAME (ASCII).
```

4. Transient exit to calling transient, using recursion register re-entry (RECUR):

```
        LDA     0,@RECUR    ; ORIGINAL FUNCTION CODE.
        MOV#    0,0,SNR     ;  IF NONE THEN
        JMP     EXIT        ;   EXIT TO CALLER.
        JMP     @LDRX       ;    RE-ENTER LOADER
                            ;     AC 0 CONTAINS FUNCTION NAME.
RECUR:  17774               ;      RECURSION REGISTER ADDRESS.
LDRX:   17775               ;       LDRX ENTRY ADDRESS.
```

5. Standard return to calling program. Uses an internal register hold AC3, which is maintained at address TOP-201:

```
        LDA     3,@AC3      ; REGISTER 3 CONTAINS DISK ERROR
                            ;  RETURN ADDRESS.
        JMP     1,3         ;   GO ONE BETTER, NORMAL RETURN.
AC3:    17577               ;    ADDRESS OF REGISTER 3 SAVE.
```

6. Complete null executable (type 1) transient example:

```
        .LOC    TOP-1340    ; TRANSIENT BUFFER BASE ADDRESS.
        START               ;  ENTRY POINT ADDRESS.
        0                   ;   NO RELOCATION REQUIRED.
START:  JMP     1,3         ;    RETURN IMMEDIATELY (NORMAL EXIT).
        .END                ;     NOT MUCH IS IT?
```

7. End of job exit. Returns directly to system control through entry at EOJ:

```
        JMP     @EOJ        ; CALL MONITOR AND EXIT.
EOJ:    17777               ;  SYSTEM EXIT (TOP-1).
```

8. Output to active console:

```
        LDA     0,CHAR      ; GET CHARACTER FROM SOMEWHERE.
        JSR     @PUTC       ;  WRITE TO CONSOLE.
        JMP     NEXT        ;   ONWARD!
PUTC:   TOP-207             ;    WRITER ENTRY POINT ADDRESS.
```

9. Input from operator console:

```
        JSR     @GETC       ; INPUT ONE CHARACTER.
        STA     0,CHAR      ;  SAVE IT SOMEPLACE.
        JMP     NEXT        ;   ONWARD!
GETC:   17570               ;    READER ENTRY POINT ADDRESS.
```

6 FILE FORMATS

MICRO utilizes three disk file types and one printer, or console, file. Source files contain ASCII coded data, while binary files contain data in 16 bits per word format. Any disk based file may be moved to or from any other disk file using the NCS utility program "COPY". COPY will also move source character record data files between disk and external media, such as magnetic tape, cards, or listing devices. Binary files cannot be moved to card or listing devices, though the "SAVE" utility program can retain a binary file on a 9 track magnetic tape.

The following sections discuss the structures and utilization of the four NCS files used during Microassemblies.

6.1 SOURCE INPUT (DISK) FILE FORMAT

Source program files, to be read through MICRO file INPT, must be resident on disk prior to initiating an assembly. The data characters on the input file may be any valid ASCII codes, though MICRO will ignore any control codes that it does not recognize. A source program may originate on any form of medium. COPY must be used to move that data from card or magnetic tape to disk. The data is organized in source record images. Each record logically begins with the first data character following an end of record code or the beginning of information point. It ends following the end of record code. A blank line may be represented by a single character, the end of record code itself.

MICRO reads INPT file data one character at a time. Each time an end of record code, which is an ASCII (015, octal) Carriage Return control code, is encountered the assembly process begins on that record. Once all statements on the record are processed MICRO returns to the disk file to retrieve the next record. When an .END statement has been recognized no further access will be made to the current INPT file. If a file has been created by COPY, that file will have been padded to its end-of-information with binary zero character codes. MICRO ignores these zero characters, and will read the INPT file to end-of-information if the .END statement is missing. Records containing more than 80 characters will be processed as 80 character records, with all excess characters being discarded.

Lower case characters are mapped into their upper case equivalents. Any ASCII control codes other than "Carriage Return" (015) will be ignored. Refer to APPENDIX A, CHARACTER SET, for additional details.

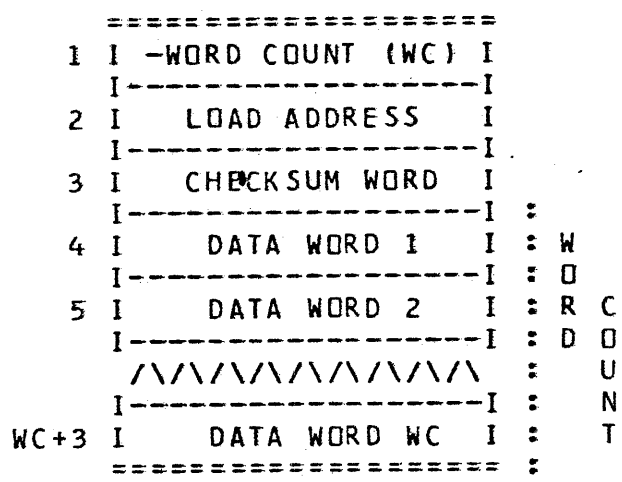
6.2 BINARY OBJECT (DISK) FILE FORMAT

MICRO object program file format consists of binary blocks, made up of 16 bit words. The BIN file is optional, and when selected must be placed on disk. It may be moved to other disk files by the COPY utility program. It may be transported, or backed up, on magnetic tape using SAVE, or on Cartridge tape using the DISCART utility. Its content may be read by the NOVA program loader "LD", by the QM/SYSTEM microprogram loader facility "QMLD", or by the cartridge tape program writer "PREP".

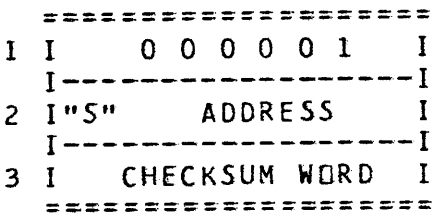
Two types of binary blocks will be generated by MICRO: Data Blocks, and Start Blocks. Data blocks contain up to 16 NOVA data words, or up to 8 microprogram (18 bit) data words. Start blocks are of a fixed length, and contain the optional program entry (start) address. A start block will cause termination of loading, or writing, when encountered by any of the object file processors. Other forms of block structures found on a BIN file will be skipped over, being terminated when a binary zero datum is read. All Data and Start blocks contain checksum words. If a checksum is found to be incorrect an error message will be written to the console.

Data blocks for NOVA type programs contain 16 bit, block load addresses. When a data block contains microprogram (18 bit) words each 18 bit word is written to the Data Block as two 16 bit words, 10 bits in the first word and 8 bits in the second. To allow for proper handling of this format the block load address must be multiplied by 2, thus restricting absolute block load addresses to a maximum value of 32,767 (77777 octal). This is a restriction on MICRO version 1.3 only. The following diagram shows the format of data and start blocks.

DATA BLOCK FORMAT



START BLOCK FORMAT



The "S" flag in the Start Block (bit position 15) indicates whether the "ADDRESS" field contains an entry point (start) address, for use in NOVA program loading. S = 0 will allow loader transfer directly to "ADDRESS". S = 1 will cause the loader to HALT after loading.

The word count of a Data Block is retained in word 1 of that block, in two's complement negative format. Word count (WC) may be from 1 to 16.

Following the Start Block, MICRO will pad the remainder of the BIN disk file with binary zero fill words. If the size of the BIN file is too small to contain the entire object program, MICRO will display an error message on the system console. The object file will be unuseable in this case, though MICRO will complete the remainder of the assembly without writing additional BIN output. The error message displayed is:

*** BINARY OUTPUT FILE FULL ***

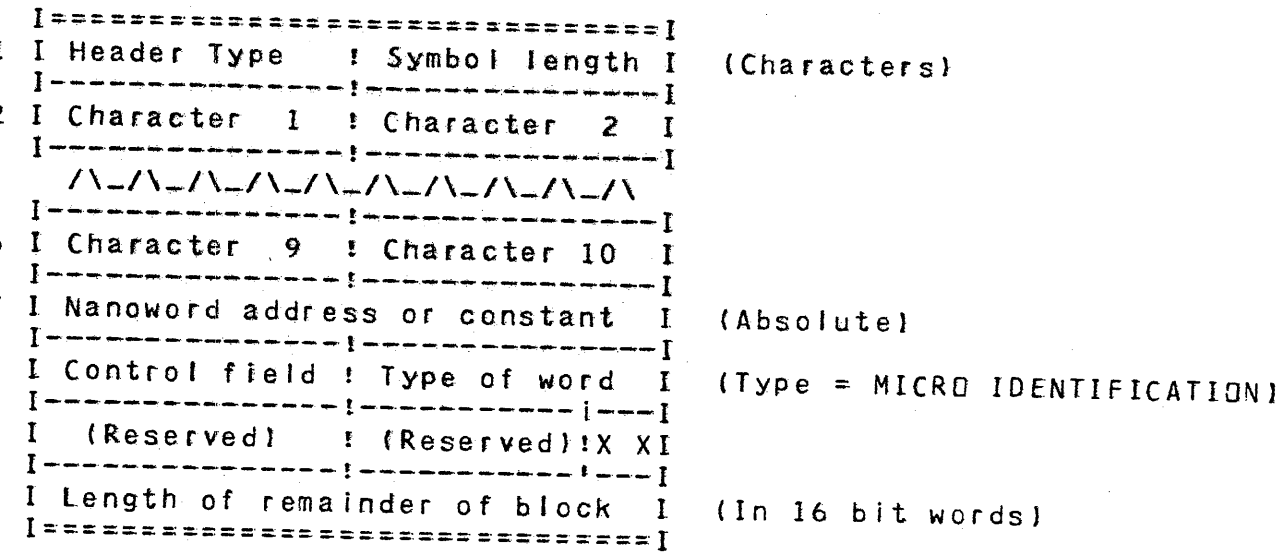
6.3 ARCHITECTURAL DEFINITION (DISK) FILE FORMAT

Each microinstruction set DEFINITION file contains a variable number of nanoword and external constant definitions. These structures are represented by binary blocks, prefixed with a 10 word (16 bit) header. Each external constant is entirely defined within a type 1 header, and is therefore always 10 words in length. Each nanoword begins with a type 2 header. A header identifies the symbolic NAME of the following word (or word group) and the number of characters in the defining symbol. Control flags and word type identification are also included. The actual assigned nanostore address is represented as the seventh header word. The last word of a header is the length of the remainder of the nanoword definition. All defined nano-primitive fields are written as 4 word record elements, immediately following the header.

An object nanoword may be completely represented in NOVA memory as a 20 word pair array, called the Nano-Logical-Array. Each record element contains the value assigned to the nano-primitive field, identified by the Nano-logical-array word Shift, Mask, and Displacement field values. The actual Nanoassembler Construction Array field number is included within each record element. Record elements are not used by MICRO, but are described in this section for completeness.

The format of Binary Output Files, produced by the version 1 Nanoassembler, is as follows. Each file begins with a type 2 header for the first real (Nanostore loadable) nanoword in the nano-module. The type of nanoword is described in the type field, shown below. End of file is identified by encountering a type 0 header word where a new header should begin (immediately following the last record element of a nanoword block). The following diagrams show the structures of the header and record elements.

NANO BINARY BLOCK HEADER



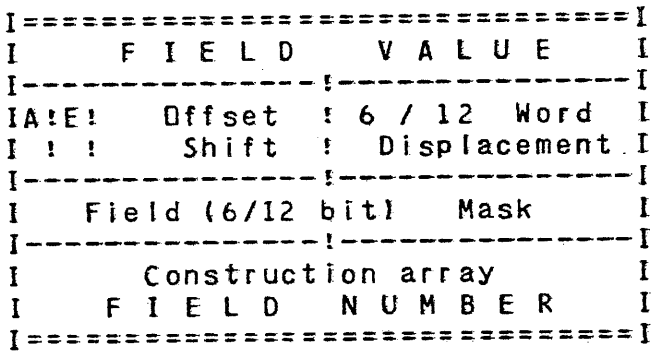
HEADER TYPES:

0 = Last Header, end of DEFINITION file.

1 = External Constant description. Always has a Length of 10 words. The actual value is in the Nanoword address field as a number which is < 2**18. The high order 2 bits of this value are in the X X portion of word 9.

2 = Nanoword Record Header. The Symbol is the name of the Nanoword, (has 0 symbol length when the word has no name) and the Characters are the first 10 ASCII symbols from the source Nanoprogram Nanoword LABEL. The address is the actual QM/SYSTEM Nanostore address. The Type field contains the microinstruction format declaration.

Fig. 6.3.B
NANOWORD RECORD ELEMENT

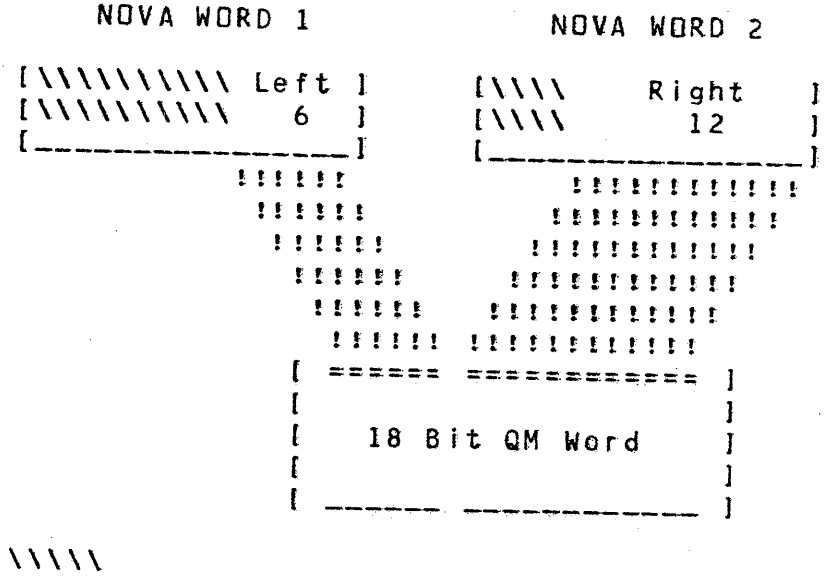


The FIELD VALUE is the binary value to be placed into the appropriate Nano-Primitive field. The Offset Shift is the amount that the Field Value should be Left Logically shifted into position for the 6/12 bit NOVA-Logical-Array word. The Field Mask is a 16 bit mask to be ANDed (after the shift) with the field value. The 6/12 Word Displacement is the indication of which word in the NOVA-Logical-Array this Field Value belongs. The Construction Array field number refers to a description table in the Nanoassembler itself.

Notes:

1. Fields "A" and "E" are internal assembler flags.
2. The real word values are set for the construction of each nanoword as 20 sets of word pairs, for NOVA implementation. Each pair represents an 18 bit QM-1 word, with the left 6 bits right justified in the first NOVA word and the right 12 bits right justified in the second NOVA word.
3. The Nanoassembler does no modular arithmetic. The Field Mask is for that purpose; it permits the nanoprogram loaders to simply insert the Nano-primitive into the appropriate field.

Fig. 6.3C NOVA-LOGICAL-ARRAY DATA WORD PAIR
NOVA (16 bit) TWO WORD FORMAT MAPPING INTO QM/SYSTEM (18 bit) FORMAT



\\\\\
\\\\\ Represents unused area.
\\\\\

6.4 SOURCE LISTING (PRINTER/CONSOLE) FILE FORMAT

The full microassembly source program listing may be output to either the line printer or system console. Each source line is divided into six fields, beginning at print character position 1. All numeric fields are displayed in 6 octal digit notation. The line formats are as follows:

```
Field:  1       2       3       4       5       6

     000000  111111 222222 333333 444444  SOURCE STATEMENT IMAGE --------->
```

MACHINE INSTRUCTION Statement:

```
     023472  620001         023473          LABEL:  B       .+1
```

Where field 1 is always the instruction address. Fields 2 and 3 will display the one or two words that will become the actual object instruction. Field 4 or 5 will contain the effective absolute storage address referenced by any instructions containing instruction-counter relative OPERANDS.

PSEUDO OPERATION statement:

```
                  000077                    MASK      =       77
  or
    010333  027102 046113 00000             CODE:     .TXT    /.BLK/
  or
    010000  777777 777777 777777 777777     FIELDS:   .BLK    9.,-1
            777777 777777 777777 777777
            777777
```

Where field 1 shows the address of the first generated word of object data, if any. Fields 2, 3, 4, and 5 will show all consecutively generated object words. If more than 4 words are generated by a single statement then the additional sets of 4 words will each be displayed on subsequent lines. The resultant value of any non-data generating statements will be displayed in field 3.

DATA statements:

```
    000100  000000 000001 000002 000003   CONSTANTS:      0, 1, 2, 3, 4, -1
            000004 777777
```

Where the format is similar to data generated by .BLK statements, above.

Error conditions detected on a statement will be displayed on the following line. Errors are flagged as single letter error codes, with the code letter usually appearing under the leftmost position of the particular element in error. When a field is incorrectly structured, or an entire statement is bad, the code letter may appear at the end of the erroneous field or statement, respectively.

Examples of errors on a statement:

```
    002310  000100 002032                  LABEL    LD   10,32   $ COMMENT 2000
    *** ERRORS FOUND IN LAST SOURCE LINE   U        U            F U
    002312  321000 000000                  NAME:    STD  10,SAVE; COMMENTS.
    *** ERRORS FOUND IN LAST SOURCE LINE                  I    O
```

Where the letter "U" indicates that the NAME is undefined, "F" is a format error, "I" is an illegal field value, and "O" indicates missing operands. Error messages are detailed in APPENDIX B, MICROASSEMBLER ERROR MESSAGES.

7 OPERATING PROCEDURE

MICRO is initiated by operator command, directly from the system console or system "COMMANDS" file. This chapter describes the preparation of data for processing by MICRO, and its invocation. The general format of the MICRO invocation command is:

```
!MA-options,DEF=definitionfiles,INPT=sourcefiles,BIN=objectfile.
```

The "!" character is the NCS command state prompt symbol, and indicates that a program name may be entered. MICRO is initiated by the program file name "MA". The parameters are explained in detail below.

7.1 PREPARATION OF SOURCE INPUT FILES

When a source program file is prepared in card format it may be copied to disk using the NCS utility program COPY. The source and object program files must be defined, using the FILES utility program, before beginning any other activity. A source program may be split into separate components for use as OVERLAYS or SEGMENTS by the various systems available. When there are common symbols or definitions to be referenced by the components of a program, these may be separated from the actual program source files. Once all source files are ready, they may be specified to MICRO as follows.

```
INPT=primaryfile+definition3+definition2+definition1,
```

Where "primaryfile" will be the actual source program from which the object code will be produced. The "definition3" through "definition1" files are all optional common symbolic definition files. These definitions will be read in numeric order, right to left. Therefore, if there are any interdependencies between these files the order of specification will be critical. The definitions will be processed only during PASS 1, and may define storage space allocations without generating any code. This may require placement of .LOC PSEUDO OPERATION statements at the beginning of each INPT file.

Example: Source input file set-up.

Definition file:

```
D       .TITL  DEFINITIONS AND COMMON SUBROUTINES
D       .LOC   100
D      /-----------------------------------------\
D     /  Common Subroutine Code and symbolic constants \
D    <   defined here for reference by primary object   >
D     \            program utilization.              /
D      \-----------------------------------------/
D  ORIGIN =    .        ; DEFINE END OF SUBROUTINES SECTION.
D       .END
```

Primary file:

```
P       .TITL  ONE PROGRAM OVERLAY SEGMENT
P       .LOC   ORIGIN   ; BEGIN AFTER SUBROUTINES SECTION.
P      /-----------------------------------------\
P     /    Main program SEGMENT, which will call on    \
P    <    elements of the common Subroutines section.   >
P     \ Subroutines may be referenced directly by name./
P      \-----------------------------------------/
P       .END   START
```

```
!COPY,*CDR,D.
!COPY,*CDR,P.
!MA,DEF=,INPT=P+D,BIN=.
```

7.2 PREPARATION OF DEFINITION FILES

DEFINITION files are produced as output from Nanoassemblies. They contain instruction set definitions as well as symbolic constant NAMES. One or more of these files must be used when performing a microassembly. The following is the format of the DEFINITION file parameter.

```
DEF=definition4+definition3+definition2+definition1,
```

Specifying no DEFINITION file is accomplished by following the DEF= with any standard NCS delimiter character, such as space " ", comma "," or period ".". The order of specification for multiple DEFINITION files is unimportant unless there may exist multiple definitions of the same symbol. In that case the symbol encountered first, from the rightmost file name, will be the symbolic value used.

Examples: Lower case lettering represents operator keyed data.
Assembly with no definitions (NOVA).

```
!copy,*tp(0),source.                <CREATE SOURCE DISK FILE>
!ma,DEF=,INPT=source,BIN=obj.       <ASSEMBLE SOURCE TO OBJ>
```

Assembly with multiple definitions.

```
!copy,*cdr,source.                  <CREATE SOURCE DISK FILE>
!ma,DEF=sysdef+usernano,INPT=source,BIN=obj.
                                    <ASSEMBLE SOURCE TO OBJ>
```

7.3 ASSEMBLER INVOCATION OPTIONS

When MICRO is initiated, by entering the letters "MA", it will produce an assembly with full source program listing on the system printer. Several options are provided for reducing or altering that listing.

The standard full listing will display all input source lines, additional lines showing error codes, and finally an operation code and symbol table list following the assembly. Options are provided that will permit deleting all listing, unconditionally, or deleting only parts of the listing.

When only lines in error are desired, or the symbol table list is not needed, MICRO may be informed of these listing options by following the MICRO initiation name with a hyphen character. The hyphen is then followed by one or more letter-option codes, as described in the following sections.

```
!MA-options,...
```

7.3.1 A - SELECT ALTERNATE DEVICE (CONSOLE)

Option "A" causes all material that would normally be transmitted to the system printer to be displayed on the system console. When a hard copy console is in use this provides an alternate listing device. In general, this option provides a quick method for displaying assembly errors, when the additional option is selected to suppress all but erroneous lines. Selecting this option also deletes listing of the operation code and symbol tables.

7.3.2 X - SUPPRESS SYMBOL TABLE LIST

The alphabetically ordered operation codes listing, and general symbol table listing, will be suppressed. Only source statement listings will be produced, if selected. Selection of option "A" will also suppress these tables.

7.3.3 N - NO LISTING, DISPLAY ERROR LINES ONLY

When "N" is specified, source statement lines will not be displayed unless that line contains one or more error conditions. Following each line in error will be an error code line, as shown in section 6.4, above. This option has no effect on any other option. Only option "L", see below, will override this option.

7.3.4 L - SUPPRESS ALL LISTABLE OUTPUT

Option "L" will override any other option specifications. No listing will be produced on any device. Error lines will also be suppressed. The only indication of error conditions occurring will be a console message produced at the end of the assembly:

```
ASSEM. ERRORS
```

7.4 BINARY OBJECT FILES

An object program file will be produced when specified as the BIN file parameter. Only one object file may be produced per assembly. Object file production may be suppressed by following the BIN parameter with an NCS delimiter character. There is no recorded identification within the object file, to distinguish between NOVA type and microprogram type object files. Care must be taken to identify these files externally, usually by file name attributes. The remainder of this section briefly describes the use of the various loaders to access object files.

```
!ma-n,DEF=sysdef,INPT=myprog:sou,BIN=myprog:obj.
```

7.4.1 MICROPROGRAM OBJECT FILES

Microprograms may usually be loaded in either of two available ways. Directly from disk object file residence, with several constraints placed on storage allocations, and from Cartridge Tape, with no constraints. The next two sections discuss the features and options available with these two methods.

7.4.1.1 "QMLD" MICROPROGRAM LOADER

"QMLD" is an NCS utility program, which utilizes the QMLDR nanoprogram loader system. One or more disk files may be merged together into a reserved Main Store area, one for each of the QM/SYSTEM storage classes. With QMLD, either object code or data may be loaded into Main Store locations 0 through 13777 (octal). Microcode and data may be loaded into Control Store locations 0 through 17777 (octal). And finally, Nanoprograms may be loaded into Nanostore from nanoword location 0 to 674. QMLD is fully described in the NCS OPERATIONS GUIDE.

7.4.1.2 "PREP" CARTRIDGE TAPE PREPARATION

The NCS utility program "PREP" provides the capability to record Cartridge Tape object program files, which can be loaded into the QM/SYSTEM memories using the standard Read-Only-Memory boot loader system. The combination of the ROM boot loader and PREP program provide the user with the ability to preset or clear areas in any of the storage regions, load program and data modules into any of the regions, and finally to initiate microprogram execution via transfer to either a nanoprogram or microprogram entry point.

The Read-Only-Memory boot loader is a Nanocode - Microcode environment, which may be interfaced through writeable Nanostore or Control Store to enhance loading capabilities. Additional information on the use of PREP and ROM is provided in the NCS OPERATIONS GUIDE.

7.4.2 "NOVA" FORMAT OBJECT FILES

A NOVA format file may be loaded directly into Main Store under NCS control using the program loader "LD". LD will read the object file and place data from the Data Blocks (see section 6.2, above) into the specified storage locations. If an entry address had been included on the .END statement of the original assembly, then control will be transferred to that address at the end of loader operation. LD may be used as follows:

```
!LD,(*)filename(-displacement).
```

Where the optional asterisk (*) will indicate to LD to suppress execution of the program even though an entry address has been provided. The negative displacement parameter is also optional, and forces all Data Blocks to be loaded lower in storage by the octal value "displacement". This allows maintenance of special NCS component programs. Once a program is

loaded in NOVA format Main Store it may be recorded on disk as an absolute system overlay program, using the following commands.

```
!$filename ENTER (Y) TO WRITE
```

If "filename" was defined, by the FILES utility program, to be an absolute NCS program file then the absolute program will be recorded on that file. This program may then be executed directly by NCS, by simply specifying the program file name as a command to NCS.

```
!filename.......
```

APPENDIX A. CHARACTER SET

Characters are stored internally as two 8 bit characters packed from left to right in a 16 bit word. The internal character set of MICRO is 7 bit ASCII right justified and zero padded within the 8 bit field. The table below gives the octal encoding of the characters, their normal associated graphic symbol and a statement of usage where appropriate.

040	space	060	0	100	@	120	P
041	!	061	1	101	A	121	Q
042	"	062	2	102	B	122	R
043	#	063	3	103	C	123	S
044	$	064	4	104	D	124	T
045	%	065	5	105	E	125	U
046	&	066	6	106	F	126	V
047	'	067	7	107	G	127	W
050	(	070	8	110	H	130	X
051	)	071	9	111	I	131	Y
052	*	072	:	112	J	132	Z
053	+	073	;	113	K	133	
054	,	074	<	114	L	134	
055	-	075	=	115	M	135	
056	.	076	>	116	N	136	
057	/	077	?	117	O	137	End of Record

Characters are extracted from the NCS file INPT one at a time, using an NCS disk utility program. MICRO assumes that the characters may have any 7 bit value and performs the following transformation:

CR (015) translated to EOR (137)
UA (136) translated to EOR (137)
140 - 177 translated to 100 - 137
Other codes 000 - 037 ignored.

Using these transformations, an input line is produced up to and including an EOR (End Of Record) resulting from either CR or UA.

APPENDIX B. MICROASSEMBLY ERROR MESSAGES

There are two classes of error messages issued by MICRO. Console messages report on global conditions, such as physical I/O errors, or assembler table errors. Statement error messages are the second class, and indicate syntactic errors encountered on the source program INPT file.

B.1 MICROASSEMBLY CONSOLE ERROR MESSAGES

1. ASSEM. ERRORS
   Indicates that one or more statement or processing errors has occurred during the assembly. This message will be produced at the end of the complete assembly process, whether or not an assembly listing was produced.

2. BAD NAME
   An illegal file name was entered during assembly invocation. The file name must be re-entered.

3. BINARY FILE SIZE = <octal value>
   Not an error message: an informative message issued only after the BIN object program file has been successfully completed. It will not be issued if a BIN file is not generated. The size is in octal notation, and represents the number of disk sectors required for the entire object program.

4. *** BINARY OUTPUT FILE FULL ***
   The BIN object file is too small to contain the entire object program. The assembly will run to completion, though the object file will be unuseable. The assembly must be re-run on a larger file.

5. * DEFINITION FILE ERROR.
   Either a physical disk error or a structural format error has been found while reading a DEFINITION file. This message will be followed by a "MALFUNCTION:" message, see (8) below.

6. DUPLICATE DEFINITION - <name>
   A symbolic NAME has been encountered more than once, on the same or separate DEF files. The NAME of the duplicate symbol is shown on the console. The value of the rightmost (first) occurrance of the NAME will be retained and used.

7. FILE NOT FOUND
   During assembler invocation, a file name either does not exist or has been incorrectly entered at the console. The name must be re-entered.

8. MALFUNCTION: 000000 111111 222222 333333
   Any physical I/O device error, or internal assembler malfunction, will cause this message to be issued. The internal NOVA registers are displayed for diagnostic purposes. The octal value found in digit string 111111 will usually be an NCS I/O error code.

9. SYMBOL TABLE OVERFLOW
Too many symbolic NAMES have been specified during the assembly. A larger version of MICRO will be necessary in order to continue.

B.2 MICROASSEMBLER STATEMENT ERROR CODES

Statement errors will be indicated on the line printed immediately below the source statement line containing the actual errors. All statement error codes are single letters, usually placed directly below the element or field in error. When an entire field specification is incorrect the error code will be placed below the field terminator. The following are descriptions of all error codes and their affects on the assembly.

A - Addressing error. The evaluated address was not within the range 0 - 377 (octal) for absolute NOVA fields, or within +200 or -177 (octal) locations from the current instruction address for location-counter relative fields.

B - BLOCK (.BLK) statement block size OPERAND value is greater than 2**18 - 1 (262,143). The block will be allocated anyway, through the micro-location-counter will usually wrap around into previously allocated areas.

C - Comma missing between required OPERANDS. Processing continues as if the comma preceeded the point of error detection.

D - NOVA Device Identification field is in error (missing or value greater than 63.) Zero is inserted in the device code field.

E - Address of object microcode exceeds 2**15 - 1 (32,767). The address generated will be modulo 32,768. This error will not be detected unless a binary file is being produced.

F - Illegal Format. Any special character code that is not recognized as a supported OPERATOR. The character is ignored, except that it will delimit any NAME or NUMBER element to which it is appended.

I - Invalid accumulator. OPERAND EXPERSSION value was greater than 3 or less than 0. Zero is inserted in the accumulator field and statement processing continues with the next field.

L - PSEUDO OPERATOR not currently supported (will be Later.)

M - Multiple definitions for the flagged symbol. The first occurrance of the symbol will be used.

N - Null text string generated by a .TXT class statement. One word of zero will be generated.

O - One or more required OPERANDS are missing. Statement processing will terminate.

P - Phase error. Symbol has different values for PASS 1 and PASS 2, or cannot be located during PASS 2. Processing continues.

Q - Operator missing within an EXPRESSION. Addition is assumed. EXPRESSION analysis continues.

R - Radix statement (.RDX) error. OPERAND EXPRESSION value is not within range, 0 to 10.

S - Invalid skip field code for NOVA type instruction. Zero is inserted into the skip field.

T - Too many OPERAND fields specified on a statement.

U - Undefined symbolic NAME, or this symbol was EQUATED to a symbol which was undefined in PASS 1. A zero value is substituted. Reference to a multiply defined NAME will also receive this error.

V - OPERAND Value overflows object field. A zero value is placed into the field, and statement processing continues.

X - Illegal symbol usage. Symbol is already an OPERATION CODE or PSEUDO OPERATOR. A zero value is substituted, and EXPRESSION processing continues.

Z - Numeric item contains digits not consistant with the current radix (base). The resulting numeric value will be the integer value of all digits preceeding the illegal digit.

APPENDIX C. PSEUDO OPERATION LIST

The following list contains a brief description of each PSEUDO OPERATION supported under the MICROASSEMBLER.

```
.RDX <Expression>                    ; SET NEW RADIX FOR NUMBER CONVERSION

.LOC <Expression>                    ; SET NEW VALUE OF LOCATION COUNTER

.BLK <Expression>                    ; RESERVE BLOCK OF MEMORY or
.BLK <Expression>,<Expression>       ; SET BLOCK OF MEMORY TO A CONSTANT VALUE

.TXTM <Expression>                   ; SET TEXT MODE

.TXT                                 ; PROCESS TEXT STREAM WITH ZERO PARITY
.TXTO                                ; PROCESS TEXT STREAM WITH ODD PARITY
.TXTE                                ; PROCESS TEXT STREAM WITH EVEN PARITY
.TXTF                                ; PROCESS TEXT STREAM WITH ONE PARITY

.XPNG                                ; DELETE PREDEFINED OP CODES

.TITL <Character String>             ; SET UP TITLE FOR DISPLAY IN PAGE HEADING

.IFN <Expression>                    ; CONDITIONAL ASSEMBLY IF NOT ZERO
.IFE <Expression>                    ; CONDITIONAL ASSEMBLY IF ZERO

.ENDC                                ; END CONDITIONAL ASSEMBLY SKIP

EJECT                                ; FORCE NEXT LINE TO BE ON A NEW PAGE

LISTON                               ; ENABLE LISTING MODE
LISTOFF                              ; DISABLE LISTING MODE

.EOT                                 ; END OF INPUT SEGMENT

.END                                 ; END OF SOURCE or
.END <Expression>                    ; END OF SOURCE WITH STARTING ADDRESS
```

Comments regarding errors, deficiencies, or omissions in this document will be appreciated. Comments should be sent in writing to:

Technical Services Manager
NANODATA CORPORATION
2457 Wehrle Drive
Williamsville, New York 14221