REFERENCE MANUAL FOR
NARP, AN ASSEMBLER FOR THE SDS 940

Roger House
Dana Angluin
Laurence P. Baker

Document No. R-32
Issued January 5, 1968
Revised November 21, 1968
Office of Secretary of Defense
Advanced Research Projects Agency
Washington, D. C. 20325

TABLE OF CONTENTS

Prefatory Note

Certain sections of the following reference manual are written in a primer-like style, especially parts of the introduction and the discussion of macros. However, it is assumed that the reader is familiar with the logical operation of general-purpose digital computers, and, in particular, is acquainted with the SDS 940 instruction set (see the SDS publication, SDS 940 Computer Reference Manual, No. 90 06 40A, August, 1966, or the Project GENIE document, SDS 930 Instructions, Document R-27, October 11, 1966).

Acknowledgment

Much of this manual is similar to the ARPAS manual (ARPAS, Reference Manual for Time-Sharing Assembler for the SDS 930, Document R-26, February 24, 1967), written by Wayne Lichtenberger, and some paragraphs are taken verbatim from the ARPAS manual.

Related Documents

1) For a precise description of the binary program output by NARP, see Project GENIE document, Format of Binary Program Input to DDT, Document R-25, January 26, 1967.

2) For a description of the implementation of NARP see Project GENIE document, Implementation of NARP, Document M-16, January 25, 1968.

1.0 Introduction

NARP (new ARPAS) is a one-pass assembler for the SDS 940 with literal, subprogram, conditional assembly, and macro facilities. The source language for NARP, primarily a one-for-one representation of machine language written in symbolic form, is very similar to that for ARPAS (another assembler for the 940), but there are notable exceptions making it necessary to do a certain amount of transliteration to convert an ARPAS program to a NARP program. No further mention will be made of ARPAS in this manual; for more details see ARPAS, Reference Manual for Time-Sharing Assembler for the SDS 930, Doc. No. R-26, February 24, 1967.

To motivate the various facilities of the assembler, the following pseudo-historical development of assembly languages is presented.

1.1 Pseudo-history of assembly languages

A program stored in the main memory of a modern computer consists of an array of tiny circular magnetic fields, some oriented clockwise, others oriented counterclockwise. Obviously, if a programmer had to think in these terms when he sat down to write a program, few problems of any complexity would be solved by computers, and the cost of keeping programmers sane would be prohibitive. To remedy this situation, utility programs called assemblers have been developed to translate programs from a symbolic form convenient for human use to the rather tedious bit patterns that the computer handles. At first these assemblers were quite primitive, little more than number converters, in fact. Thus, for example:

Tag	Opcode	Address
∅	76	∅∅4∅∅
∅	55	∅∅4∅1
∅	35	∅∅4∅2

would be converted into three computer instructions which would add together the contents of cells 4ØØ and 4Ø1 and place the result in cell 4Ø2. An assembler for doing this type of conversion is trivial to construct.

After a time, some irritated programmer who could never remember the numerical value of the operation "load the A register with the contents of a cell of memory" decided that it would not be too difficult to write a more sophisticated assembler which would allow him to write a short mnemonic word in place of the number representing the hardware operation. Thus, the sequence of instructions shown above became:

```
Ø LDA ØØ4ØØ
Ø ADD ØØ4Ø1
Ø STA ØØ4Ø2
```

This innovation cost something, however, namely the assembler had to be more clever. But not much more clever. The programmer in charge of the assembler simply added a table to the assembler which consisted of all the mnemonic operation names (opcodes) and an associated number, namely the numerical value of the opcode. When a mnemonic name, say 'ADD', was encountered by the assembler during the conversion of a program, the opcode table was scanned until the mnemonic name was found; then the associated numerical value (in this case, 55) was used to form the instruction. Within a month, no programmer could tell you the numerical value of XMA.

In a more established field, the innovation of these mnemonic names would have been quite enough for many years and many theoretical papers. However, programmers are an irritable lot, and furthermore, are noted for their ability to get rid of sources of irritation, either by writing more clever programs or by asking the engineers to refrain from making such awkward machines. And the use of numbers to represent addresses in memory was a large source of irritation. To see this we need another example:

```
Ø CLA
Ø LDX ØØ4ØØ
2 STA ØØ5Ø7
Ø BRX ØØ3ØØ
```

Assuming cell 4ØØ contains -7, this sequence stores zeroes in cells 5ØØ through 5Ø6 provided that the sequence is loaded in memory so that the STA instruction is in cell 3ØØ (otherwise, the BRX instruction would have to be modified). This was the crux of the problem: Once a program was written, it could only run from a fixed place in memory and could only operate on fixed cells in memory. This was especially awkward when a program was changed, since inserting an instruction anywhere in a program would generally require changes in many, many addresses. One day a clever programmer saw that this problem could be handled by a generalization of the scheme used to handle opcodes, namely, let the programmer use symbolic names (symbols) for addresses and have the assembler build a table of these symbols as they are defined and then later distribute the numerical values associated with the symbols as they are used. Thus the example becomes:

```
            CLA
            LDX    TABLEN
    LOOP    STA    TABEND,2
            BRX    LOOP
```

(Note that at the same time the programmer decided to move the tag field to after the address field (simply for the sake of readability) and to even dispense with it entirely in case it was zero.) The assembler now has two tables, the fixed opcode table with predefined names in it, and a symbol table which is initially empty. There is also a special cell in the assembler called the location counter (LC) which keeps track of how many cells of program have been assembled; LC is initially zero. There is another complication: In the above example, when the symbol TABLEN is encountered, it may not be defined yet, so the assembler doesn't know what numerical value to replace it with. There are several clever ways to get around this problem, but the most obvious is to have the assembler process the program to be assembled twice. Thus, the first time the assembler scans the program it is mainly interested in the symbol definitions in the left margin (a symbol used to represent a memory address is called a label). In our example, when LOOP is encountered, it is stored in the symbol table and given the value 2 (because it is preceded by two cells; remember that LC keeps track of this). At the end of pass 1, all symbols defined in the program are in the symbol table with numerical values corresponding to their addresses in the memory. So when pass 2 begins, the symbol table is used exactly as the opcode table is used, namely, when, for example, LOOP is encountered in the BRX instruction above, it is looked up in the symbol table and replaced by the value 2. If the program should later be changed, for example to

```
            CLA
            LDB     EIGHT
            LDX     TABLEN
    LOOP    STP     TABEND,2
            EAX     1,2
            BRX     LOOP
```

then the assembler will automatically fix up LOOP to have the value 3 (because of the inserted LDB instruction) and will convert BRX LOOP to BRX 3 instead of to BRX 2 as before. Thus, the programmer can forget about adjusting a lot of numerical addresses and let the assembler do the work of assigning new values to the symbols and distributing them to the points where the symbols are used. In addition to the greater flexibility achieved, symbols with mnemonic value can be used to make the program more readable.

The use of symbols to stand for numerical values which are computed by the assembler and not the programmer is the basic characteristic of all assembly languages. Its inception was a fundamental breakthrough in machine language programming,dispensing with much dullness and tedium. And a new breed of programmer was born: the assembler-writer. To justify his existence, the assembler-writer began to add all sorts of bells and whistles to his products; the primary ones are discussed in the next section (with reference to NARP).

1.2 Assembly languages: some basic constituents and concepts

Times: assembly time: when a program in symbolic form is converted by an assembler to binary (relocatable) program form.

load time: when a binary program is converted by a loader to actual machine language in the main memory of the computer.

run time: when the loaded program is executed.

source program —assembler→ binary program —loader→ object program

Expressions: The idea of using a symbol to stand for an address is generalized to allow an arithmetic expression (possibly containing symbols) to stand for an address. Thus, some calculations can be performed at assembly time rather than at run time, making programs more efficient.

Literals: Rather than writing LDA M1 and somewhere else defining M1 to be a cell containing -1, the literal capability allows the programmer to write the contents of a cell in the address field instead of the address of a cell. To indicate this, the expression is preceded by '='. The assembler automatically assigns a cell for the value of the expression (at the end of the program):

```
            CLA
            LDB     =8
            LDX     =-16*2
LOOP        STP     TABBEG+16*2,2
            EAX     1,2
            BRX     LOOP
```

Relocation: A relocatable program is one in which memory locations have been computed relative to the first word or origin of the program. A loader (for this assembler, DDT) can then place the assembled program into core beginning at whatever location may be specified at load time. Placement of the program involves a small calculation. For example, if a memory reference is to the nth word of a program, and if the program is loaded beginning at location k, the loader must transform the reference into absolute location n+k. This calculation should not be done to each word of a program since some machine instructions (shifts, for example) do not refer to memory locations. It is therefore necessary to inform the loader whether or not to relocate the address for each word of the program. Relocation information is determined automatically by the assembler and transmitted as a relocation factor (rfactor). Constants or data may similarly require relocation, the difference here being that the relocation calculation should apply to all 24 bits of the 940 word, not just to the address field. The assembler accounts for this difference automatically.

Subprograms and external symbols: Programs often become quite large or fall into logical divisions which are almost independent. In either case it is convenient to break them into pieces and assemble (and even debug) them separately. Separately assembled parts of the same program are called subprograms (or packages). Before a program assembled in pieces as subprograms can be run it is necessary to load the pieces into memory and link them. The symbols used in a given subprogram are generally local to that subprogram. Subprograms do, however, need to refer to symbols defined in other subprograms. The linking process takes care of such cross-references. Symbols used for it are called external symbols.

Directives: A directive (pseudo-opcode is a message to the assembler serving to change the assembly process in some way. Directives are also used to create data:

```
         LIST
MESSAGE  TEXT   'THIS IS A PIECE OF TEXT'
START    LDA    ALPHA
```

The LIST directive will cause the program to be listed during assembly, while the TEXT directive will cause the following text to be stored in memory, four characters to a word.

Conditional assembly: It is frequently desirable to permit the assembler to either assemble or skip a block of statements depending on the value of an expression at assembly time; this is called conditional assembly. With this facility, totally different object programs can be generated, depending on the values of a few parameters.

Macros: A macro is a block of text defined somewhere in the program and given a name. Later references to this name cause the reference to be replaced by the block of text. Thus, the macro facility can be thought of as an abbreviation or shorthand notation for one or more assembly language statements. The macro facility is more powerful than this, however, since a macro may have formal arguments which are replaced by actual arguments when the macro is called.

One-pass assembly: Instead of processing a source program twice as was described above (section 1.1), NARP accomplishes the same task in one scan over the source program. The method used is rather complex and is described elsewhere. (Implementation of NARP, Doc. M-16)

2.0 Basic constituents of NARP

2.1 Character set

All the characters listed in Appendix B have meaning in NARP except for '?' and '\'. The following classification of the character set is useful:

letter:	A-Z
octal digit:	0-7
digit:	0-9
alphanumeric character:	letter or digit or colon
terminator:	, ; blank CR (denotes carriage return)
operator:	! # % & * + - / < = > @ ↑
delimiter:	" $ ' ( ) [ ] . ←

The multiple-blank character (135_8) may appear anywhere that a blank is allowed. All characters with values greater than 77_8 are ignored except for multiple-blank character (135_8) and carriage return (155_8).

2.2 Statements and format

The logical unit of input to NARP is the statement, a sequence of characters terminated by a semi-colon or a carriage return. There are five kinds of statements:

1. empty: A statement may consist of no characters at all, or only of blank characters.
2. comment: If the very first character of a statement is an asterisk, then the entire statement is treated as a comment containing information for a human reader. Such statements generate no output.

The format for the next three kinds of statements is split into four fields:

label field: This field is used primarily for symbol definition; it begins with the first character of the statement and ends on the first non-alphanumeric character (usually a blank).

opcode field: This field contains a directive name, a macro name, or an instruction (i.e., any opcode other than a directive or macro). The field begins with the first non-blank character after the label field and terminates on the first non-alphanumeric character; legal terminators for this field are blank, asterisk, semi-colon, and carriage return.

operand field: The operand for an instruction, macro, or directive appears in this field, it begins with the first non-blank character following the opcode field and terminates on the first blank, semi-colon, or carriage return. Note that a statement may terminate before the operand field.

comment field: This field contains no information for NARP but may be used to help clarify a program for a human reader. The field starts with the first non-blank character after the operand field (or after the opcode field if the opcode takes no operand) and ends on a semi-colon or carriage return.

Now we continue describing the kinds of statements:

3. instruction: If the opcode field of a statement does not contain a directive name or a macro name, then the statement is an instruction. An instruction usually has an expression as an operand and generates a single machine word of program. See section 3 for a detailed description of instructions.

4. directive: If a directive name appears in the opcode field, then it is a directive statement. The action of each directive is unique and thus each one is described separately (in section 4).

5. macro: A macro name in the opcode field of a statement indicates that the body of text associated with the macro name should be processed (see section 5).

Example of various kinds of statements:

```
* FOLLOWING ARE TWO DIRECTIVES (MACRO,ENDM) WHICH DEFINE
* THE MACRO SKAP
SKAP MACRO;    SKA =4B7;   ENDM
```

```
*    NOW SKAP IS CALLED:
          LDA ALPHA
          SKAP;     BRU BAD        IF NEGATIVE THEN ERROR
OKAY      ADD BETA     NOW A=ALPHA+BETA;     BRU GOOD
```

In subsequent sections the details of instructions, directives, and macros will be explained, but first some basic constituents and concepts common to all of these statements will be discussed.

2.3 Symbols, numbers, and string constants

Any string of alphanumeric characters not forming a number is a symbol, but only the first six characters distinguish the symbol (thus Q12345 is the same symbol as Q123456). Note that a symbol may begin with a digit, and that a colon is treated as a letter (as a matter of good programming practice, colons should be used rarely in symbols, although they are often useful in macros and other obscure places to avoid conflicts with other names). In the next section the definition and the rfactors of symbols are discussed.

A number is any one of the following:

a) A string of digits
b) A string of digits followed by the letter 'D'
c) A string of octal digits followed by the letter 'B'
d) A string of octal digits followed by the letter 'B' followed by a single digit.

A D-suffix indicates the number is decimal, whereas a B-suffix indicates an octal number. If there is no suffix, then the current radix is used to interpret the number (the current radix is initially 10 but it may be changed by the OCT and DEC directives). If the digit 8 or 9 is encountered in an octal number, then an error message is typed. If the value of a number exceeds 2^{23}-1 overflow results; NARP does not check for this condition, and in general it should be avoided. A B-suffix followed by a digit indicates an octal scaling; thus, 74B3=74ØØØB.

Examples:

symbols: START 1M CALCULATE 14D2 14B10

numbers: 14 18D 773B 777B5 13B9

A string constant is one of the following:

a) A string of 1 to 3 characters enclosed in double quotes (").

b) A string of 1 to 4 characters enclosed in single quotes (').

In the first case the characters are considered to be 8 bits each (thus only 3 can be stored in one machine word), while in the second case they are considered to be 6 bits each. In both cases, strings of less than the maximum length (3 or 4, as the case may be) are right-justified. Thus

'A' = '␣␣␣A' = "A" = "␣␣A"

where ␣ denotes a blank. If a string constant is too long, then an error message is typed and only the first 3 (or 4) characters are taken. Normally string constants are not very useful in address computation, but are most often used as literals:

```
LDA     WORD
SKE     ='GO'
BRU     STOP
```

Both numbers and string constants are absolute, i.e., their rfactor is zero.

2.4 Symbol definitions

Since NARP is a one-pass assembler, the statement that a symbol or expression is "defined" usually means that it is defined at that instant and not somewhere later in the program. Thus, assuming ALPHA is defined nowhere else, the following

```
BETA    EQU   ALPHA
ALPHA   BSS   3
```

is an error because the EQU directive demands a defined operand and ALPHA is not defined until the next statement. This convention is not strictly adhered to, however, since sometimes the statement "XYZ is not defined" will mean that XYZ is defined nowhere in the program.

A symbol is defined in one of two ways: by appearing as a label or by being assigned a value with an EQU directive (or equivalently, by being assigned a value by NARG, NCHR, EXT (see below), or by being used in the increment list of a RPT or CRPT statement). This latter sort of symbol is called equated.

Labels: If a symbol appears in the label field of an instruction (or in the label field of some directives) then it is defined with the current value of the location counter (rfactor=1). If the symbol is already defined, either as a label or as an equated symbol, the error message '(Symbol) REDEFINED' is typed and the old definition is completely replaced by the new one.

Equated symbols: These symbols are usually defined by EQU, getting the value of the expression in the operand field of the EQU directive. This expression must be defined and have an rfactor in the range [-15,15]. If the symbol has been previously defined as a label, then the error message '(Symbol) REDEFINED' is typed and the old definition is completely replaced by the new one; if the symbol has already been defined as an equated symbol, then no error message is given, and the old value and rfactor are replaced by the new ones. Thus, an equated symbol can be defined over and over again, getting a new value each time.

A defined symbol is always local, and may also be external. If a symbol in package A is to be referred to from package B, it must be declared external in package A. This is done in one of the following ways:

Declared external by $: If a label or equated symbol is preceded by a $ when it is defined, then it is declared external.

```
$LABEL1   LDA   ALPHA
LABEL2    STA   BETA     LABEL2 IS LOCAL ONLY
$GAMMA    EQU   DELTA
```

Declared external by the EXT directive: There are two cases:

i) EXT has no operand: The symbol in the label field is declared external; it may have already been declared external or may even have a $ preceding it.

ii) EXT has an operand: This case is treated exactly like the case: $label EQU operand.

Certain symbols are pre-defined in NARP, i.e., they already have values when an assembly begins and need not be defined by the programmer:

:ZERO: This is a relocatable zero (i.e., value = 0, rfactor = 1).

:LC: This symbol is initially zero (rfactor=1) and remains so until the END directive is encountered and all literals are output, at which time it gets the value of the location counter. See the description of FREEZE for a discussion of the use of this symbol.

* Syntactically this is not a symbol, but semantically it acts like one. At any given moment, * has the value of the location counter (rfactor=1), and can thus be used to avoid creating a lot of local labels.

Thus

```
                CLA;  LDX LENGTH
        LOOP    STA TABLE,2;  BRX LOOP
```

can be written as

```
                CLA;   LDX LENGTH;   STA TABLE,2;   BRX *-1
```

If a given symbol is referred to in a program, but is not defined when the END directive is encountered then it is assumed that this symbol is defined as external in some other package. Whether this is the case cannot be determined until the various packages have been loaded by DDT. Such symbols are called "undefined symbols" or "external symbol references." It is possible to perform arithmetic upon them (e.g., LDA UNDEF+1); an expression in post-fix Polish form will be transmitted to DDT.

2.5 Expressions and literals

Loosely speaking, an expression is a sequence of constants and symbols connected by operators. Examples:

```
100-2*ABC/[ALPHA+BETA]
GAMMA
F>=Q
```

Following is the formal description (in Backus normal form) of a NARP expression:

```
<primary>::=<number>|<string constant>|<symbol>|*|[<expr>]
<expr>::=<primary>|<unary operator> <expr>|<expr> <binary operator> <expr>
<expression>::=<expr>|<literal operator> <expr>

<binary operator>::=↑|*|/|+|-|<|<=|=|#|>=|>|&|!|%
<unary operator>::=+|-|@
<literal operator>::= =
```

Notice that the literal operator is rather special, only being allowed to appear once in a given expression, and only as the first character of the expression. Literals are discussed in greater detail below.

The value of an expression is obtained by applying the operators to the values of the constants and symbols, evaluating from left to right except when this order is interrupted by the precedence of the operators or by square brackets* ([,]); the result is interpreted as a 24-bit signed integer. The following table describes the various operators and lists their precedences (the higher the precedence, the tighter the operator binds its operands):

---

*not parentheses!

Operator	Precedence	Comment	
↑	6	exponentiation; exponent must be ≥ 0	
*	5	multiplication	
/	5	integer division	
+ (u)	4	unary plus	
- (u)	4	negation (arithmetic)	
+	4	addition	
-	4	subtraction	
<	3	less than	
<=	3	less than or equal to	result of operation is
=	3	equal to	0 if relation is false,
#	3	not equal to	otherwise 1
>=	3	greater than or equal to	
>	3	greater than	
@ (u)	2	logical not	
&	1	logical and	logical operation
!	0	logical or	applied to all
%	0	logical exclusive or	24 bits

The rfactor of an expression is computed at the same time the value is computed. There are constraints, however, on the rfactors of the operands of certain operators, as shown in the table below: (Note: R1 is a symbol with an rfactor of 1, R2 is a symbol with an rfactor of 2).

operator	relocation factor(s) of operand(s)	relocation factor of result	examples
↑ & ! % /	all operands absolute	absolute	2↑4=16, R1↑1(error) 7&3=3, 6&R1(error) 4/2=2, R1/1(error)
*	at least one rfactor must be absolute, the other is arbitrary	found by multiplying the value of the absolute operand times the rfactor of the other operand	3*R2 has rfactor of 6, R1*R1(error)
< <= = # >= >	arbitrary relocation factors, but must be equal	absolute	R1=R1 is true R2>R1(error)
+ - (unary and binary)	arbitrary rfactors	found by applying operator to the relocation factors of the operands	R1+R2 has relocation factor of 3

The final rfactor of an expression must be in the range [-8191, 8191].

If an expression contains an undefined symbol or if it is a literal, then the entire expression is undefined.

Although a literal is a special kind of expression, it is often convenient to think of it as a quite separate entity. The use of literals is discussed below.

Programmers frequently write such things as

```
LDA    FIVE
```

where FIVE is the name of a cell containing the constant 5. The programmer must remember to include the datum FIVE in his program somewhere. This can be avoided by the use of a literal.

```
LDA    =5
```

will automatically produce a location containing the correct constant in the program. Such a construct is called a literal. When a literal is encountered, the assembler first evaluates the expression and looks up its value in a table of literals constructed for each subprogram. If it is not found in the table, the value is placed there. In any case the literal itself is replaced by the location of its value in the literal table. At the end of assembly the literal table is placed after the sub-program.

The following are examples of literals:

```
=10    =4B6    =ABC*20-DEF/12    ='HELP'
=2>AB    (This is a conditional literal. Its value will
          be 1 or 0 depending on whether 2>AB at assembly
          time.)
```

Some programmers tend to forget that the literal table follows the subprogram. This could be harmful if the program ended with the declaration of a large array using the statement

```
ARRAY    BSS    1
```

It is not strictly correct to do this, but some programmers attempt it anyway on the theory that all they want to do is to name the first cell of the array. The above statement will do that, of course, but only one cell will be reserved for the array. If any literals were used in the subprogram, they would be placed in the following cells which now fall into the array. This is, of course, an error. Other than this exception, the programmers need not concern himself with the locations of the literals.

3.0 Instructions

There are three different syntactical forms of instruction statements, depending on the class of the instruction in the opcode field: (In the following, syntactical elements enclosed in square brackets are optional; they may or may not be present.)

```
class Ø:  [[$]label]  opcode[*]  [operand[,tag]  [comment]]
class 1:  [[$]label]  opcode[*]  [comment]
class 2:  [[$]label]  opcode[*]  operand[,tag]  [comment]
```

Each of the syntactical elements is discussed below:

$: A label preceded by a dollar sign is declared external (see section 2.4).

label : The label is defined with the current value of the location counter (rfactor=1).

opcode : The opcode must be either an instruction which is already defined or a number. If it is a number, then the value (mod 2^9) of the number is placed in bØ-b8 (bit Ø through bit 8) of the instruction, and it is treated as a class Ø opcode (i.e., operand optional).

* : If an asterisk follows immediately after the opcode then b9 (the indirect bit) of the instruction is set.

operand: The operand is an expression which may or may not be defined and which has any rfactor. The expression may be preceded by '/' or '←' (or both in any order); these characters cause the following bits to be set:

/	b1	(index bit)
←	b9	(indirect bit)

Thus:

LDA /VECTOR	is the same as	LDA VECTOR,2
STA ←POINTER	is the same as	STA* POINTER
LDA ←/COMPLX	is the same as	LDA* COMPLX,2

tag : The tag is an expression which must be defined and absolute. Its value (mod 2^3) is placed in bØ-b2 of the instruction.

comment: The comment does not affect the instruction generated; it may be listed.

In addition to its class, a given opcode is designated as being either a shift instruction or a non-shift instruction. This has nothing to do with whether the action of the instruction involves shifting, but is simply a way of distinguishing between two kinds of instructions. For non-shift instructions, operands are computed mod 2^{14}, while for shift instructions there are two possibilities:

a) If the indirect bit is set by '*' or '←', then the value of the opcode is trimmed so that b10-b23 are zero, and then the instruction is treated as if it were a non-shift instruction.

b) If the indirect bit is not set as above, then the operand is computed mod 2^9; it must be defined and absolute.

4.0 Directives

There are many directives in NARP; although some of them are similar, each in general has its own syntax. Following is a concise summary:

Class	Directive	Use or Function	Section
Mnemonic for instructions:	COPY	Mnemonic for RCH	4.4
Data generation :	DATA	Generate data	4.5
	ASC	Generate text (3 characters per word)	4.1
	TEXT	Generate text (4 characters per word)	4.23
Value declaration :	EQU	Equate a symbol to a value	4.9
	EXT	Define a symbol as external	4.10
	NARG	Number of arguments	5.5
	NCHR	Number of characters	5.5
	OPD	Define an opcode	4.17
	POPD	Define a programmed operator	4.19
Assembler control :	BES	Block ending symbol	4.2
	BSS	Block starting symbol	4.3
	END	End of assembly	4.8
	DEC	Interpret integers as decimal	4.6
	OCT	Interpret integers as octal	4.16
	FRGT	Do not output a specific symbol	4.12
	FRGTOP	Suppress output of opcode	4.125
	IDENT	Identification of a package	4.13

Class	Directive	Use or Function	Section
	DELSYM	Do not output any symbols	4.7
	RELORG	Assemble relative with absolute origin	4.20
	RETREL	Return to relocatable assembly	4.22
	FREEZE	Preserve symbols, opcodes, and macros	4.11
Output and listing control :	LIST	Set listing controls	4.14
	NOLIST	Reset listing controls	4.15
	PAGE	Begin a new page on the listing	4.18
	REM	Type out remark	4.21
Conditional assembly and macros :	IF	Begin if body	5.1
	ELSF	Alternative if body	5.1
	ELSE	Alternative if body	5.1
	ENDF	End if body	5.1
	RPT	Begin repeat body	5.2
	CRPT	Begin conditional repeat body	5.2
	ENDR	End repeat body	5.2
	MACRO	Begin macro body	5.4
	LMACRO	Alternative to MACRO	5.4
	ENDM	End macro body	5.4

In the remainder of this section, all directives listed above except for those associated with conditional assembly and macros are described.

4.1 ASC Generate text (3 characters per word)

[[$]label] ASC string [comment]

This directive creates a string of 8-bit characters stored 3 to a word. The string starts in the leftmost character of a word and takes up as many words as needed; if the last word is not filled up completely with characters from the string, then the right end of the word is filled out with blanks. If a label appears, its value is the address of the first word of the string. The syntactical element "string" is usually any sequence of characters (not containing a single quote) surrounded by single quotes. However, the first character encountered after 'ASC' is used as the string delimiter (of course, blanks and semi-colons cannot be used as string delimiters).

Examples:

```
       ASC   'NO SINGLE QUOTES, HERE IS A SEMI-COLON:;'
$ALPHA ASC   $HERE IS A SINGLE QUOTE: '$
```

4.2 BES Block ending symbol

[[$]label] BES expression [comment]

The location counter is incremented by the value of the expression in the operand field and then the label (if present) is given the new value of the location counter. Thus, in effect, a block of words is reserved and the label addresses the first word after the block. The expression must be defined and absolute. This directive is most often used in conjunction with the BRX instruction, as in the following loop for adding together the elements of an array:

```
            LDX   =-LENGTH;   CLA;   ADD ARRAY,2
            BRX   *-1;    STA   RESULT;   HLT
      ARRAY BES   LENGTH
```

4.3 BSS Block starting symbol

[[$]label] BSS expression [comment]

This directive does exactly the same thing as BES except that the label (if present) is defined before the location counter is changed. Thus, the label addresses the first word of the reserved block. It should be noted that the expression for both BES and BSS may have a negative value, in which case the location counter is decremented.

4.4 COPY Mnemonic for RCH

[[$]label] COPY $s_1, s_2, s_3, \ldots$ [comment]

(where s_i are symbols from a special set associated with the COPY directive)

The COPY directive produces an RCH instruction. It takes in its operand field a series of special symbols, each standing for a bit in the address field of the instruction. The bits selected by a given choice of symbols are merged together to form the address. For example, instead of using the instruction CAB (04600004), one could write COPY AB. The special symbol AB has the value 00000004.

The advantage of the directive is that unusual combinations of bits in the address field--those for which there exist normally no operation codes--may be created quite naturally. The special symbols are mnemonics for the functions of the various bits. Moreover, these symbols have this special meaning only when used with this directive; there is no restriction on their use either as symbols or opcodes elsewhere in a program. The symbols are:

Symbol	Bit	Function
A	23	Clear A
B	22	Clear B
AB	21	Copy (A) → B
BA	20	Copy (B) → A
BX	19	Copy (B) → X
XB	18	Copy (X) → B
E	17	Bits 15-23 (exponent part) only
XA	16	Copy (X) → A
AX	15	Copy (A) → X
N	14	Copy -(A) → A (negate A)
X	1	Clear X

To exchange the contents of the B and X registers, negate A, and only for bits 15-23 of all registers, one would write

```
COPY   BX,XB,N,E
```

4.5 DATA Generate data

[[$]label] DATA $e_1, e_2, e_3, \ldots$ [comment]

The DATA directive is used to produce data in programs. Each expression in the operand field is evaluated and the 24-bit values assigned to increasing memory locations. One or more expressions may be present. The label is assigned to the location of the first expression. The effect of this directive is to create a list of data, the first word of which may be labeled.

Since the expressions are not restricted in any way, any type of data can be created with this directive. For example:

```
        DATA    100,-217B,START,AB*2/DEF,'NUTS',5
```

creates six words.

4.6 DEC Interpret integers as decimal

DEC [comment]

The radix for integers is set to ten so that all following integers (except those with a B-suffix) are interpreted as decimal. When an assembly begins the radix is initialized to ten, so DEC need never be used unless the OCT directive is used.

4.7 DELSYM Do not output any symbols

DELSYM [comment]

If DELSYM appears anywhere in a program being assembled, the symbol table and opcode definitions will not be output by NARP when the END directive is encountered. The main purpose of this directive is to shorten the object code generated by the assembler, especially when the symbols are not going to be needed later by DDT.

4.8 END End of assembly

END [comment]

When this directive is encountered the assembly terminates. If the LIST directive has been used then various information may be listed, for example undefined symbols.

4.9 EQU Equate a symbol to a value

[$]symbol EQU expression [comment]

The symbol is defined with the value of the expression; if the symbol is already defined, its value and rfactor are changed. The expression must be defined and must have an rfactor in the range [-15,15]. If the symbol has been declared external before or if it has been forgotten (using FRGT) then EQU preserves this information. Thus

```
$ALPHA    EQU   4
ALPHA     EQU   3
```

will cause ALPHA to be declared external but with a value of three at the end of the assembly (provided ALPHA is not changed again before the END directive). See section 2.4 for more discussion of EQU.

4.10 EXT Define a symbol as external

[$]symbol   EXT   [expression   [comment]]

This directive is used to declare symbols as external. See section 2.4 for a discussion of the various cases.

4.11 FREEZE Preserve symbols, opcodes, and macros

FREEZE [comment]

Sometimes subprograms share definitions of symbols, opcodes, and macros. It is possible to cause the assembler to take note of the current contents of its symbol and opcode tables and the currently defined macros and include them in future assemblies, eliminating the need for including copies of this information in every subprogram's source language.

When the FREEZE directive is used, the current table boundaries for symbols and opcodes and the storage area for macros is noted and saved away for later use. These tables may then continue to expand during the current assembly. (A separate subprogram may be used to make these definitions; it will then end with FREEZE; END.) The next assembly may then be started with the table boundaries returned to what they were when FREEZE was last executed. This is done by entering the assembler at its "continue" entry point, i.e., by typing

@CONTINUE NARP.

Note that the assembler cannot be released (i.e., another subsystem like QED or DDT cannot be used) without losing the frozen information.

In conjunction with the FREEZE directive, the predefined symbol :LC: is useful, especially when writing large re-entrant programs. Following is a three-package program using FREEZE and :LC:.

```
P1      IDENT
        <definitions of macros, opcodes, and global equated
             symbols>
        <definition of working storage (i.e., read-write
             memory)>
        FREEZE
        END

P2      IDENT
        BSS      :LC:-:ZERO:
        <read-only code>
        END
```

```
P3      IDENT
        BSS     :LC:-:ZERO:
        <read-only code>
        END
```

The FREEZE directive at the end of P1 preserves all the definitions in this package so they can be referenced in packages P2 amd P3. By including the definitions of all the working storage cells in the preserved definitions, these symbols need not be declared as external. Also, it makes "external" arithmetic on these symbols possible in P2 and P3, and it reduces the number of undefined symbols printed at the end of an assembly. Packages P2 and P3 start with the rather peculiar looking BSS in order to set the location counter so that references between the packages will be correct. This is the main purpose of :LC:, it saves the final value of the location counter from the previous package for use by the current package. In order for this scheme to work, <u>all three</u> packages must be loaded at the same location, usually 0 for large re-entrant programs.

Assume ALPHA is a symbol defined in P1. Unless some special action is taken, ALPHA will be output to DDT three times, once at the end of P1, once at the end of P2, and once at the end of P3. To avoid this, all symbol and opcode definitions are marked after they have been output once so that they won't be output again.

4.12 FRGT Do not output a specific symbol

FRGT $s_1, s_2, \ldots$ [comment]

The symbols s_i (which must have been previously defined) are not output to DDT. FRGT is especially useful in situations where symbols have been used in macro expansions or conditional assemblies, and have meaning only at assembly time. When DDT is later used, memory locations are sometimes printed out in terms of these meaningless symbols. It is desirable to be able to keep these symbols from being delivered to DDT, hence the FRGT directive.

4.125 FRGTOP Forget selected opcodes

FRGTOP $s_1, s_2, \ldots$ [comment]

The s_i must be opcodes. The specified opcodes are marked as forgotten and will not be output to DDT. Since DDT knows in advance about the standard instruction set (e.g., LDA, BRS, CIO), FRGTOP on such opcodes has no effect. It follows that the chief use of FRGTOP will be to suppress output of opcodes generated by OPD and POPD.

FRGTOP does not take a label.

4.13 IDENT Identification of a package

symbol IDENT [comment]

The symbol in the label field is delivered to DDT as a special identification record. DDT uses the IDENT name in conjunction with its treatment of local symbols: in the event of a name conflict between local symbols in two different subprograms, DDT resolves the ambiguity by allowing the user to concatenate the preceding IDENT name with the symbol in question. Also, during an assembly the first six characters of the symbol followed by the word 'IDENT' are typed on the teletype to show the user what package is being assembled. The progress of an assembly can be followed by placing IDENT's at various points in the package.

4.135 LIBEXT Specify library symbol

Symbol LIBEXT [comment]

This directive causes "symbol" to be output to the binary file, marked as a special "library-symbol." The resulting binary file must then be mauled by a library-making program before it will be intelligible to the loader in DDT.

The library-maker takes a binary file and moves all of the library-symbols to the beginning of the program, and puts the result on a file as a "library-program." When a "library-file" (which contains one or more library-programs) is loaded into DDT, the loader scans the list of library symbols before each library-program. If any of them is currently undefined (i.e., referenced but not defined in previously loaded programs), the associated library-program is loaded normally; otherwise, it is not loaded.

For example, one could write a sine and cosine library program:

```
SIN       LIBEXT

*SINE ROUTINE:  ANGLE IN RADIANS
$SIN      ZRO     SINX
          (sine routine code)

COS       LIBEXT

*COSINE ROUTINE:  ANGLE IN RADIANS
$COS      ZRO     COSX
          (cosine routine code)

          END
```

Assemble it with NARP and use the library-maker to put it on a library-file as a library-program. Then, if either "SIN" or "COS" is undefined when the library-file is loaded, both the sine and cosine subroutines will be loaded, and the symbols "SIN" and "COS" defined as the entry points of the routines (respectively). (If one desired to have them load independently, each subroutine could be made into a separate library-program.)

(Note: The library-program is loaded normally once the decision to load it has been made; thus, undefined library-symbols will only be defined and linked in previously-loaded programs if they are defined and made external in the library-program in the usual fashion (as in the example).)

4.14 LIST Set listing controls

4.15 NOLIST Reset listing controls

$\left\{\begin{matrix}\text{LIST}\\\text{NOLIST}\end{matrix}\right\}$ [s_1, ... [comment]]

There are various booleans which control the format in which statements are listed (certain fields and/or certain kinds of statements may be suppressed, or listed selectively). The user is allowed to set (or reset) these booleans via the LIST (or NOLIST) command. Each of the s_i may be one of the following special symbols:

s_i Set (or reset)	What is (or is not) listed
LCT	the current value of the location counter, in octal
SLCT	the symbolic address of the current value of the location counter
VAL	the value of the statement, if it is one of the directives: EQU, NCHR, NARG, IF, ELSF. (in octal)
SRC	the symbolic source code
COM	the comment field of a statement, a comment statement
CALL	macro and RPT calls
DEF	MACRO and RPT definitions
EXP	macro and RPT expansions
SKIF	the skipped parts of an IF statement
EXT	external symbol references (at the end of the assembly

In addition, s_i may be "ALL", which will cause all of the booleans in the table to be set (or reset).

If a LIST (or NOLIST) directive is encountered for which no arguments (s_i) have been specified, NARP will begin (or cease) listing statements on the LISTING FILE (the teletype, in case no other listing file is specified when the assembly is begun) according to the current settings of the listing booleans. Including "GO" among the arguments for a LIST (or NOLIST) will have the same effect.

When NARP is called, the listing booleans are initialized as follows:

Set: LCT, VAL, SRC, COM, CALL, DEF, EXP, EXT
RESET: SLCT, SKIF

and NARP is in its "no list" state, i.e., listing will not be started unless (and until) the program initiates it with a LIST directive.

Examples of the LIST directive:

```
        NOLIST     ALL        Resets all format booleans
        LIST       SRC, GO    Sets SRC boolean and starts listing.
```

(only the source code will be listed)

Examples of listing format:

LCT	SLCT	VAL	SRC			COM
00117	(A)	3	A	EQU	6/2	(SET A)
00117	(HERE)		HERE	LDA	A*B,2	
00120	(HERE+1)			CLB		

4.16 OCT Interpret integers as octal

OCT [comment]

The radix for integers is set to eight so that all following integers (except those with a D-suffix) are interpreted as octal.

4.17 OPD Define an opcode

symbol   OPD   value[,class[,shift kludge]]

The symbol in the label field is defined as an opcode with a value equal to the first expression in the operand field. All expressions in the operand field must be defined and absolute; if an optional expression does not appear then the value 0 is assumed.

value	:	computed mod 2^{24} (see important note below)
class	:	must have a value of 0,1, or 2:
		0 - the opcode may or may not have an operand
		1 - the opcode does not take an operand
		2 - the opcode requires an operand
shift kludge:		must have a value of 0 or 1:
		0 - non-shift instruction (see section 3)
		1 - shift instruction (see section 3)

Note: Although an opcode that takes operands can be defined with bits b10-b23 set, the user must be careful of what he is doing. In particular, if such an opcode appears in an instruction which contains a literal or an undefined value then bits b10-b23 of the opcode are set to zero.

If the symbol in the label field is already defined as an opcode then the old definition is lost.

Examples:

```
ADD    OPD    055B5,2
CLA    OPD    04600001B,1
RCY    OPD    0662B4,2,1
NOP    OPD    020B5
```

4.19 POPD Define a programmed operator

symbol   POPD   value[,class[,shift kludge]]

This directive does exactly what OPD does with one addition: The instruction BRU* is placed in the memory location whose address is in b2-b8 of the value given to the symbol (this address must be in the range [100B, 177B]). Thus

```
MIN     POPD    100B5,2
LMIN    SKG*    0            THE CALL 'MIN ALPHA' WILL
        BRR     0            CAUSE THE MINIMUM OF
        LDA*    0            A-REG AND ALPHA TO BE
        BRR     0            LEFT IN A-REG.
```

will cause BRU LMIN to be loaded in word 100B.

4.20 RELORG Assemble relative with absolute origin

RELORG expression [comment]

On occasion it is desirable to assemble in the midst of otherwise normal program a batch of code which, although loaded in core in one position, is destined to run from another position in memory. (It will first be moved there in a block.) This is particularly useful when preparing program overlays. The expression in the operand field (which must be absolute and defined) denotes an origin in memory. The following occurs when the directive is encountered:

a.) The current value of the location counter is saved, and in its place is put the absolute origin (i.e., the value of the expression). This fact is not revealed to DDT, however, so during loading the next instruction assembled will be placed in the next memory cell available as if nothing had happened.

b.) The mode of assembly is switched to absolute, i.e., all symbols defined in terms of the location counter will be absolute.

It is possible to restore normal relocatable assembly (see section 4.22).

As an example of the use of RELORG, consider a program beginning with RELORG 300B. The assembler's output represents an absolute program whose origin is 00300_8, but which can be loaded anywhere using DDT in the usual fashion. Of course, before executing the program it will be necessary to move it to location 00300_8.

As another example, consider the following use of RELORG and RETREL:

```
<normal relocatable program>
RELORG 100B
<absolute program with origin at 100B>
RELORG 200B
<absolute program with origin at 200B>
```

```
RETREL
<normal relocatable program>
RELORG 300B
<absolute program with origin at 300B>
END
```

4.21 REM Type out remark

REM text

This directive causes the text in its operand and comment fields to be typed out either on the teletype or whatever file has been designated as the text file (see section 6.2). This typeout occurs regardless of what listing controls are set. The directive may be used for a variety of purposes: It may inform the user of the progress of assembly; it may give him instructions on what to do next (this might be especially nice for complicated assemblies); it might announce the last date the source language was updated; or it might be used within complex macros to show which argument substrings have been created during expansion of a highly nested macro (for debugging purposes).

4.22 RETREL Return to relocatable assembly

RETREL [comment]

This directive is used when it is desired to return to relocatable assembly after having done a RELORG. It is not necessary to use RETREL unless one desires more relocatable program. An example of the use of RETREL is shown in section 4.20. The effects of RETREL are

a.) to restore the location counter to the value it would have had if the RELORG (s) had never appeared, and

b.) to return the assembly to relocatable mode so that labels are no longer absolute.

4.23 TEXT Generate text (4 character per word)

[[$]label] TEXT string [comment]

This directive is exactly the same as ASC (see section 4.1) except that characters are taken as six bits each and are stored four to a word.

5.0 Conditional assemblies and macros

5.1 IF, ELSF, ELSE, and ENDF   If statements

It is frequently desirable to permit the assembler either to assemble or to skip blocks of statements, depending on the value of an expression at assembly time. This is primarily what is meant by conditional assembly. In NARP, conditional assembly is done by using either an if statement or a repeat statement.

The format of an if statement is

```
IF       expression       [comment]
< if body >
ENDF     [comment]
```

The if body is any block of NARP statements, in particular, it may contain directives of the form

```
ELSF      expression      [comment]
```

and

```
ELSE      [comment]
```

If the operand of IF is true, then the block of code up to the matching ENDF (or ELSF or ELSE) is processed; otherwise, it is skipped. The values for true and false are:

true : value of expression $> \emptyset$

false : value of expression $\leq \emptyset$

Examples:

```
IF      1>Ø
LDA     ALPHA  }  processed
STA     BETA   }
ENDF

IF      Ø
LDA     GAMMA  }  skipped
STA     DELTA  }
ENDF
```

Often there are more than two alternatives, so the ELSF directive is used in the if body. When ELSF is encountered while skipping a block of statements, its operand is evaluated (just as for IF) to decide whether to process the block following the ELSF.

Examples:

```
IF      Ø > 1
LDA     ALPHA        skipped
ELSF    1 > Ø
LDA     BETA         processed
ENDF

IF      Ø > 1
LDA     ALPHA        skipped
ELSF    Ø > 1
LDA     BETA         skipped
ENDF

IF      1 > Ø
LDA     ALPHA        processed
ELSF    1 > Ø
LDA     BETA         skipped
ENDF

IF      Ø > 1
LDA     ALPHA        skipped
ELSF    1 > Ø
LDA     BETA         processed
ELSF    1 > Ø
LDA     GAMMA        skipped
ENDF
```

From the last two examples above it should be clear that either no blocks are processed or precisely one is; thus, as soon as one block is processed, all following blocks are skipped regardless of whether the ELSF expressions are true.

An ELSE directive is equivalent to an ELSF directive with a true expression.

Example:

```
IF      Ø > 1
LDA     ALPHA          skipped
ELSE
LDA     BETA           processed
ENDF
```

As a more general example, consider the following:

```
IF      e1
< body 1 >
ELSF    e2
< body 2 >
ELSF    e3
< body 3 >
ELSE
< body 4 >
ENDF
```

There are four possibilities:

```
a)  e1 > Ø              :  process body 1, skip the other three
b)  e1 ≤ Ø, e2 > Ø      :  process body 2, skip the other three
c)  e1 ≤ Ø, e2 ≤ Ø,
      e3 > Ø            :  process body 3, skip the other three
d)  e1 ≤ Ø, e2 ≤ Ø,
      e3 ≤ Ø            :  process body 4, skip the other three
```

The bodies between the IF, ELSF, ELSE, and ENDF directives may contain arbitrary NARP statements, in particular they may contain other if statements. This nesting of if statements may go to any level.

When evaluating the expression in the operand field of IF or ELSF, all undefined symbols are treated as if they were defined with value -1. These expressions must be absolute.

5.2 RPT, CRPT, and ENDR   Repeat statements

A repeat statement is a means of processing the same text many times. The format is

```
[[$]label]     RPT     expression[,increment list]     [comment]
               < repeat body >
               ENDR     [comment]
```

The value of the RPT operand (which must be defined and absolute) determines how many times the repeat body will be processed, while the increment list (described below) is a mechanism to allow the values of various symbols to be changed each time the repeat body is processed.

Example:

```
          ABC     RPT      4
                  DATA     0
                  ENDR
```

This is equivalent to

```
          ABC     DATA     0
                  DATA     0
                  DATA     0
                  DATA     0
```

An increment list has the form (s=el[,e2])...,(s=el[,e2]) where s stands for a symbol and el and e2 denote expressions (which must be absolute; undefined symbols are treated as if they were defined with the value -1). Before the repeat body is processed for the first time, each symbol in the list is given the value of its associated el. Thereafter, each symbol is incremented by the value of its associated e2 just before the repeat body is processed. If e2 is missing, the value 1 is assumed. There is no limit on the number of elements that may appear in an increment list.

Example:

```
RPT     3,(I=4)(J=0,-1)
DATA    I
DATA    J*I+1
ENDR
```

This results in code equivalent to the following:

```
DATA    4
DATA    0*4+1    =1
DATA    5
DATA    -1*5+1   =-4
DATA    6
DATA    -2*6+1   =-11
```

There is another format for RPT:

[[$]label]   RPT   (s=e1[,e2],e3)[increment list]   [comment]

In this case, the number of times the repeat body is processed is determined by the construct (s=e1[,e2],e3). This is the same as an increment list except that it includes a third expression (which must be absolute; all undefined symbols are treated as if they were defined with the value -1), namely a bound on the value of the symbol. As soon as the bound is passed, processing of the repeat body stops. In the example above, the same effect could have been achieved by writing the head of the repeat statement as

```
RPT     (J=0,-1,-2)(I=4)
```

or as

```
RPT     (I=4,6)(J=0,-1)
```

Note that the bound does not have to be positive or greater than the initial value of the symbol being incremented; the algorithm for determining when the bound has been passed is given below.

Occasionally one wishes to perform an indefinite number of repeats, terminating on an obscure condition determined in the course of the repeat operation. The conditional repeat directive, CRPT, serves this function. Its effect is like that of RPT (and its repeat body is also closed off with an ENDR) except that instead of giving a number of repeats, its associated expression is evaluated just prior to each processing of the repeat body to determine whether to process the block. As for IF, > 0 means true, ≤ 0 means false; the expression must be defined and absolute each time it is evaluated. The form is

[[$]label] CRPT expression[,increment list] [comment]

For example, one may write

    CRPT   X>Y

or

    CRPT   STOP,(X=1,2)(Y=-3)

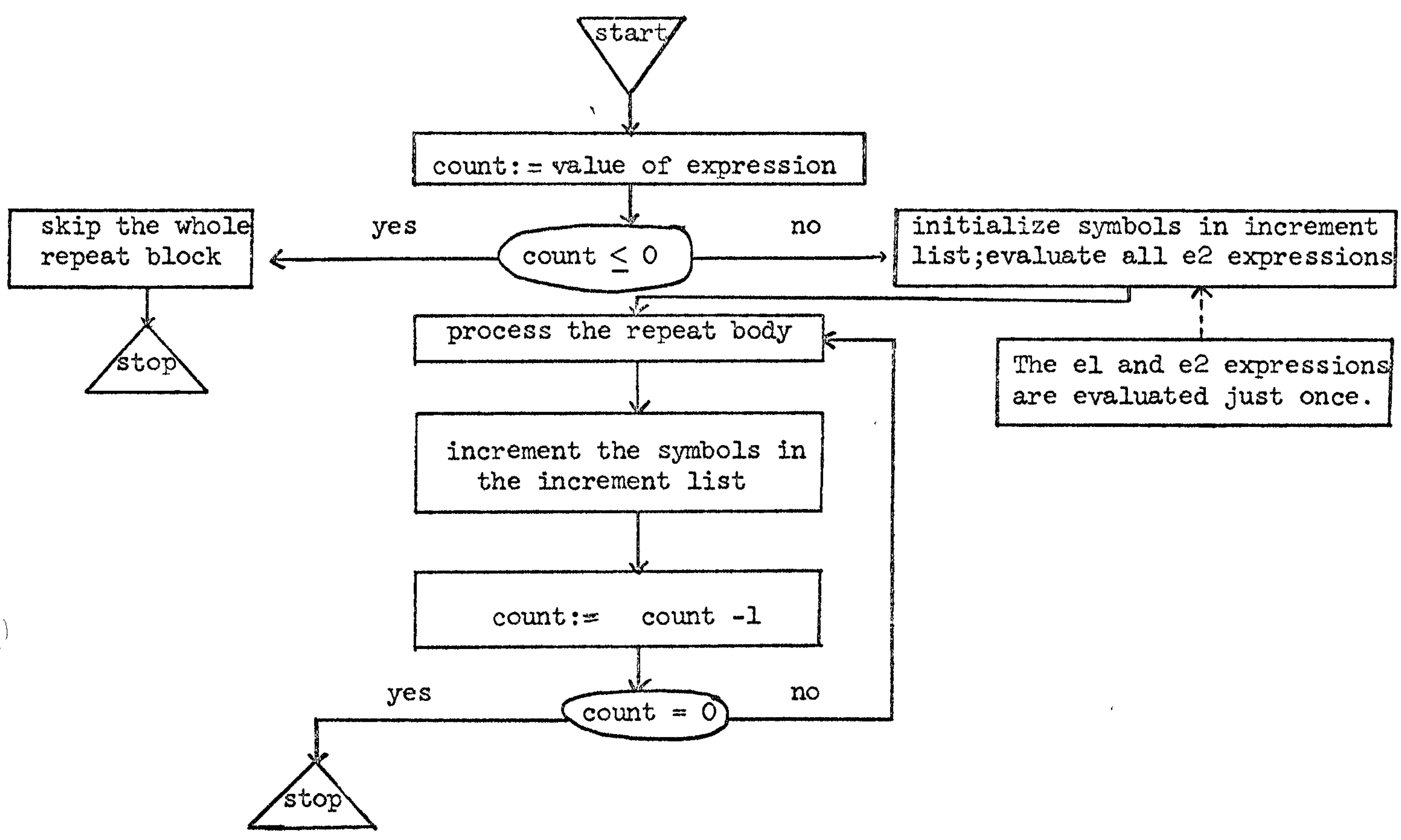
Note that the statement

    CRPT   10

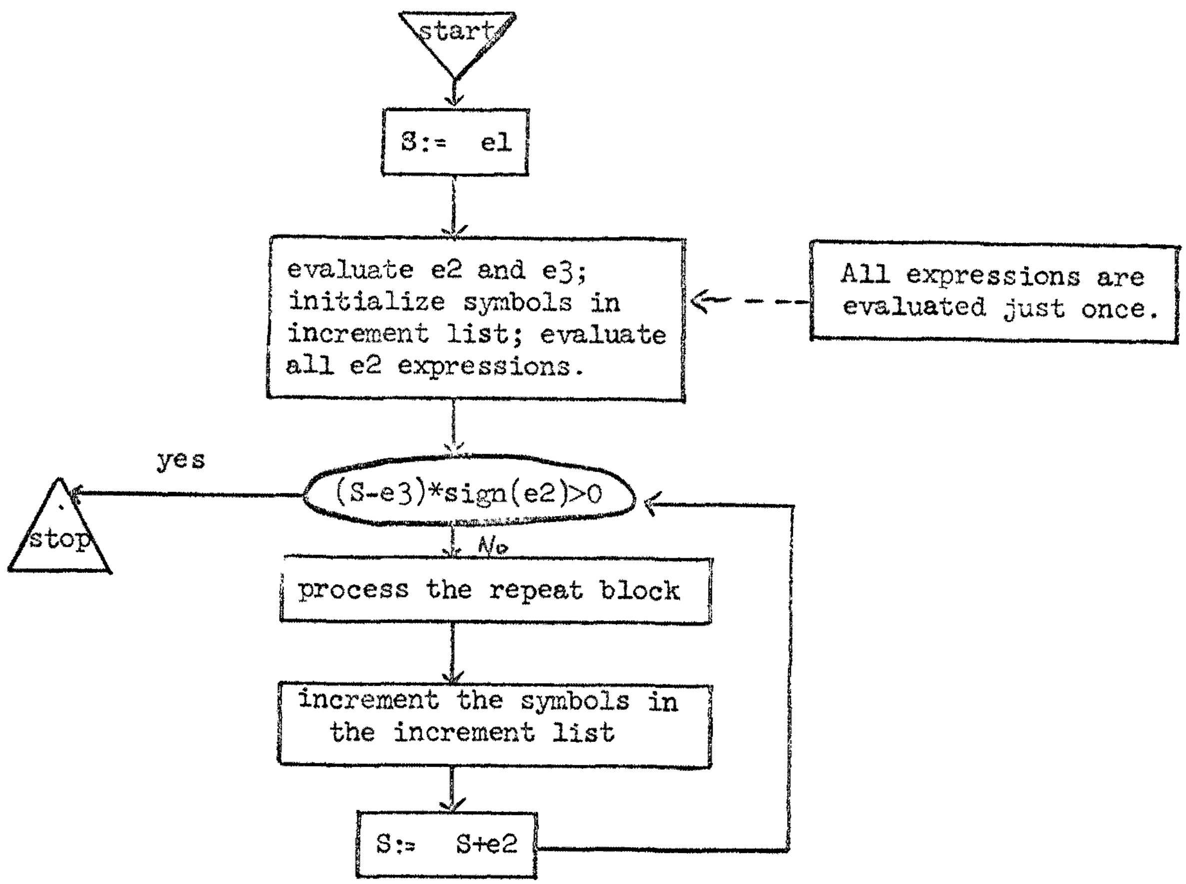
will cause an infinite number of repeats.

The following flowcharts describe precisely the actions of the various repeat options:

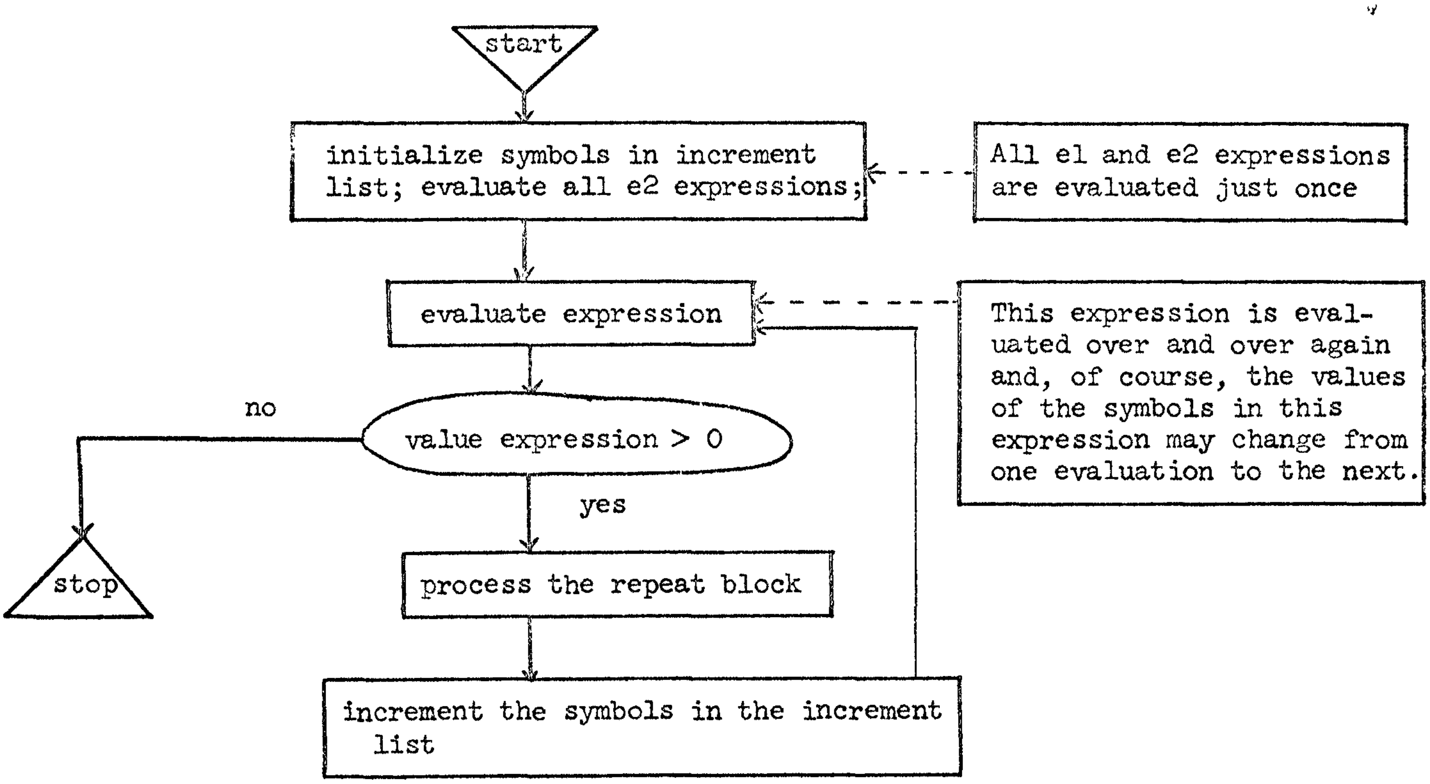
    RPT   expression[,increment list]



RPT (S=e1[,e2],e3)[increment list]



CRPT expression[,increment list]



The contents of a repeat body may contain any NARP code, in particular it may contain other repeat statements; the nesting of repeat statements may go to any level.

5.3 Introduction to macros

On the simplest level a macro name may be thought of as an abbreviation or shorthand notation for one or more assembly language statements. In this respect it is like an opcode in that an opcode is the name of a machine command and a macro name is the name of a sequence of assembly language statements.

The 940 has an instruction for skipping if the contents of a specified location are negative, but there is no instruction for skipping if the accumulator is negative. The instruction SKA (skip if memory and the accumulator do not compare ones) will serve when used with a cell whose contents mask off all but the sign bit. The meaning of SKA when used with such an operand is "skip if A is positive". Thus a programmer writes

```
        SKA     =4B7
        BRU     NEGCAS        NEGATIVE CASE
```

However, it is more than likely the case that the programmer wants to skip if the accumulator is negative. Then he must write

```
        SKA     =4B7
        BRU     *+2
        BRU     POSCAS        POSITIVE CASE
```

Both of these situations are awkward in terms of assembly language programming.

But we have in effect just developed simple conventions for doing the operations SKAP and SKAN (skip if accumulator positive or negative). Define these operations as macros:

```
SKAP    MACRO
        SKA     =4B7
        ENDM


SKAN    MACRO
        SKA     =4B7
        BRU     *+2
        ENDM
```

Now, more in keeping with the operations he had in mind, the

Programmer may write

```
A22     SKAN
        BRU     POSCAS
```

The advantages of being able to use SKAP and SKAN should be apparent. The amount of code written in the course of a program is reduced; this in itself tends to reduce errors. A greater advantage is that SKAP and SKAN are more indicative of the action that the programmer had in mind, so that programs written in this way tend to be easier to read. Note, incidentally, that a label may be used in conjunction with a macro. Labels used in this way are usually treated like labels on instructions; they are assigned the current value of the location counter. This will be discussed in more detail later.

Before discussing more complicated uses of macros, some additional vocabulary should be established. A macro is an arbitrary sequence of assembly language statements together with a symbolic name. During assembly, the macro is stored in an area of memory called the string storage. Macros are created (or, as is more frequently said, defined) by giving a name and the associated sequence of statements. The name and the beginning of the sequence of statements are designated by the MACRO directive:

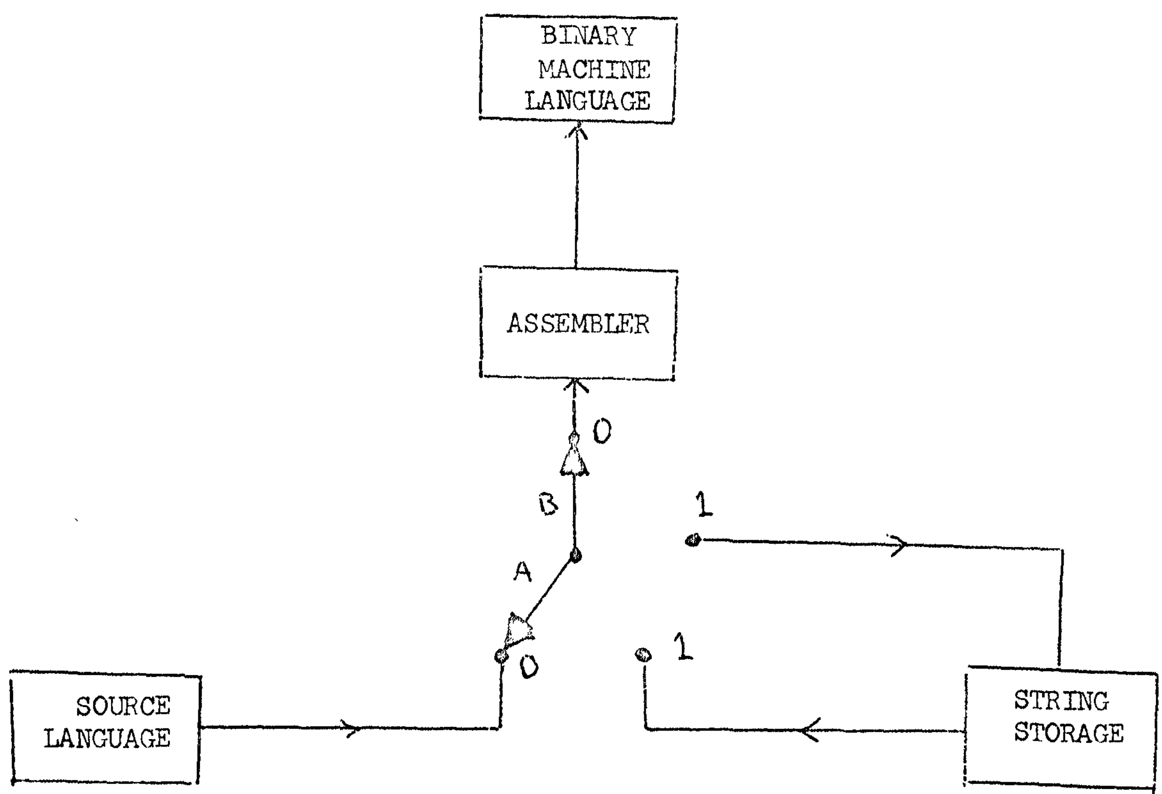
```
name    MACRO
          .
          .
        ENDM
```

The end of the sequence of statements is indicated by the ENDM directive.

Refer to figure 1. When the assembler encounters a MACRO directive, switch B is thrown to position 1 so that the macro is simply copied into the string storage; note that the assembler does no normal processing but simply copies the source language. When the ENDM terminating the macro definition is encountered, switch B is put back to position Ø and the assembler goes on processing as usual.

It is possible that within a macro definition other definitions

Figure 1: Information Flow During Macro Processing



A	B	Effect
0	0	normal assembly
0	1	macro definition
1	0	macro expansion
1	1	macro definition during macro expansion

may be embedded. The macro defining machinery counts the occurrences of the MACRO directive and matches them against the occurrences of ENDM. Thus switch B is actually placed back in position 0 only when the ENDM matching the first MACRO is encountered. Therefore, MACRO and ENDM are opening and closing brackets around a segment of source language. Structures like the following are possible:

```
name 1     MACRO
name 2     MACRO
name 3     MACRO
           ENDM
name 4     MACRO
           ENDM
           ENDM
name 5     MACRO
           ENDM
           ENDM
```

The utility of this structure will not be discussed here. Use of this feature of imbedded definitions should in fact be kept to a minimum since the implementation of this assembler is such that it uses large amounts of string storage in this case. What is important, however, is an understanding of <u>when</u> the various macros are defined. In particular, when name 1 is being defined, name 2, 3, etc., are <u>not</u> defined; they are merely copied into string storage. Name2, for example, will not be defined until name1 is expanded. (It should be noted that macros, like opcodes, may be redefined.)

The use of a macro name in the opcode field of a statement is referred to as a <u>call</u>. The assembler, upon encountering a macro call, moves switch A to position 1 (see figure 1). Input to the assembler from the original source file temporarily stops and comes instead from string storage. During this period the macro is said to be undergoing <u>expansion</u>. It is clear that a macro must be defined before it is called.

An expanding macro may include other macro calls, and these, in turn, may call still others. In fact, macros may even call themselves; this is called <u>recursion</u>. Examples of the recursive use of macros are given later. When a new macro expansion begins within a macro expansion, information about the progress of the current expansion is saved. Successive macro calls cause similar information to be saved. At the end of each expansion the information about each previous expansion is restored. When the final expansion terminates, switch A is placed back in position 0, and input is again taken from the source file.

Now let us carry our example one step further. One might argue that the action of skipping is itself awkward. It might be preferable to write macros BRAP and BRAN (branch to specified location if contents of accumulator are positive or negative). How is one to do this? The location to which the branch should go is not known when the macro is defined, in fact, different locations will be used from call to call. The macro processor, therefore, must enable the programmer to provide some of the information for the macro expansion at call time. This is done by permitting dummy arguments in macro definitions to be replaced by arguments (i.e., arbitrary substrings) supplied at call time. Each dummy argument is referred to in the macro definition by a subscripted symbol. This symbol or dummy name is given in the operand field of the MACRO directive.

Let us define the macro BRAP:

```
BRAP        MACRO       LABEL
            SKAN
            BRU         LABEL(1)
            ENDM
```

When called by the statement 'BRAP   POSCAS', the macro will expand to

```
            SKA         =4B7
            BRU         *+2
            BRU         POSCAS
```

Note that BRAP was defined in terms of another macro, SKAN. Also note that as defined BRAP was intended to take only one argument; other macros may use more than one argument.

The macro CBE (compare and branch if equal) takes two arguments. The first argument is the location of a cell to be compared for equality with the accumulator; the second is a branch location in case of equality. The definition is

```
CBE     MACRO   D
        SKE     D(1)
        BRU     *+2
        BRU     D(2)
        ENDM
```

When CBE is called by the statement

```
CBE     =21B,EQLOC
```

the statements generated will be

```
SKE     =21B
BRU     *+2
BRU     EQLOC
```

Note that in the macro call, the arguments are separated by commas.

The following sections describe macro definitions and calls in more detail.

5.4 MACRO, LMACRO, and ENDM Macro definition

The form of a macro definition is:

```
            ┌ MACRO ┐
name       {   or    }      [dummy[,generated,expression]]  [comment]
            └ LMACRO┘
```

where name, generated, and dummy are all symbols, and expression is an expression.

LMACRO is completely equivalent to MACRO except that if name is defined as a macro with MACRO the construct

```
label     name     arguments
```

will automatically cause label to be defined as the current value of the location counter, whereas if name were defined with LMACRO this automatic definition of label would not occur.

Some details of the definition

If generated appears, it should not be the same symbol as dummy, and neither of them should be "MACRO", "LMACRO", or "ENDM."

If name is already defined as an opcode, the old definition is completely replaced by the new.

If the MACRO (or LMACRO) directive has no operand, then name is defined as an opcode that takes no operands. Otherwise, name becomes an opcode that may or may not take an operand.

Whole-line comments (lines beginning with *) in the macro body are not saved in string storage as part of the macro definition, but comments following instructions are. Thus, it behooves the programmer to avoid the latter, as they eat string storage.

When a macro body is placed in string storage, superfluous blanks are removed. Thus, any contiguous string of blanks is compressed to one blank with the following exceptions:

a) Blanks enclosed in single quotes (') are not compressed.

b) Blanks enclosed in double quotes (") are not compressed.

c) Blanks enclosed in parentheses are not compressed. In this use, the nesting of parentheses is taken into account, but a parenthesis between single or double quotes is not considered as part of the nesting structure.

In most cases the programmer need not worry about these conventions, although there are times when he may get pinched. For example, if

```
ASC     %A␣␣␣B%
```

appears in a macro definition, it will be expanded as

```
ASC     %A␣B%
```

To avoid such problems use

```
ASC     'A␣␣␣B'
```

5.4.1 Dummy arguments

The dummy argument specified as an operand of the MACRO directive may appear anywhere in the macro body, followed by a subscript. At call time the subscript is evaluated and its value is used to select the appropriate argument supplied in the call. Before describing the various kinds of dummy arguments a few conventions are needed:

a) In the following, "argument" will refer to the character string as given in the macro call after possible enclosing parentheses have been removed (see section 5.6 for the format of argument strings).

b) The number of arguments supplied by the call is n ($n \geq 0$).

c) The number of characters in argument ei is n(ei).

d) The structure ei for i an integer stands for an expression. (However, its value stands for some argument usually, so ei will be used somewhat ambiguously to stand for an expression or the value of an expression.) The first argument in a call is numbered 1.

e) The dummy argument is assumed to be "D".

With the above in mind, we consider the three forms of dummy arguments:

1) D(e1)

This expands to argument e1 (which may be the null string), where $0 \leq e1 \leq n$. (If e1 = 0 then D(e1) expands to the label field of the macro call; see section 5.6.)

Special notation: D() = D(1)

2) D(e1,e2)

If e1 > e2 then this expands to the null string (range of values of e1 and e2 is arbitrary), otherwise, this expands to argument e1 through e2, where $0 \leq e1 \leq e2 \leq n$, with each argument enclosed in parentheses and a comma inserted between each argument. For example, D(3,3) = (D(3)).

Special notation: D(,) = D(1,n)
D(,e1) = D(1,e1)
D(e1,) = D(e1,n)

3) D(e1$e2,e3)

In all cases, $0 \leq e1 \leq n$ must be true. If $e2 > e3$ then this expands to the null string (range of values of e2 and e3 is arbitrary), otherwise, it expands to characters e2 through e3 of argument e1, counting the first character of an argument as character 1. If either e2 or e3 lies outside the argument, then the nearest boundary is chosen. To be more precise, before using e2 and e3 to select the piece of argument e1 that is desired, the following transformation is made:

```
e2:=    max (1,e2);       e3:=    max (1,e3);
e2:=    min (n(e1), e2);   e3:=    min (n(e1),e3);
```

If argument e1 is the null string, then the dummy argument expands to the null string regardless of the values of e2 and e3.

Special notations:

```
D(e1$,) = D(e1$1, n(e1)) = D(e1)
D(e1$,e2) = D(e1$1,e2)
D(e1$e2,) = D(e1$e2,n(e1))
D(e1$e2) = D(e1$e2,e2)
D(e1$) = D(e1$1) = D(e1$1,1)
```

In any of the six forms mentioned above, e1 may be missing; if so, 1 is assumed. E.g., D($) = D(1$1,1).

A general rule which will help in remembering what the special notations mean is the following: "Whenever an expression is missing from a form, the value 1 is assumed unless the expression is missing from a place where an upper bound is expected (as in D(3,) or D(3$2,), in which case the largest 'reasonable' value is assumed."

In any of the above three cases, if an expression which designates an argument is out of range, then an error message is typed and argument 0 is taken.

Following is an example of the various forms of dummy arguments:

Macro definition:

```
XAMPLE      MACRO     D
            D(2)      D()        D(0)
            ASC       'D(2,4)'
            TEXT      'D(4,)'    D(-3,-4)      NULL STRING
            ASC       'D(1$3,4)'
            ASC       'D(2$-3,18)'
            ENDM
```

Macro call:

```
BETA        XAMPLE    ALPHA,ADD,GAMMA,DELTA
```

Macro expansion:

```
BETA        ADD       ALPHA        BETA
            ASC       '(GAMMA),(DELTA)'
            TEXT      '(DELTA)'   NULL STRING
            ASC       'PH'
            ASC       'ADD'
```

5.4.2 Generated symbols

A macro should not, of course, have in its definition an instruction having a label. Successive calls of the macro would produce a multiply-defined symbol. Sometimes, however, it is convenient to put a label on an instruction within a macro. There are at least two ways of doing this. The first involves transmitting the label as a macro argument when it is called. This is most reasonable in many cases; it is in fact often desirable so that the programmer can control the label being defined and can refer to it elsewhere in the program.

However, situations do arise in which the label is used purely for reasons local to the macro and will not be referred to elsewhere. In cases like this it is desirable to allow for the automatic creation of labels so that the programmer is freed from worrying about this task. This may be done by means of the generated symbol.

A generated symbol name may be declared when a macro is defined, specifying the name and the maximum number of generated symbols which will be encountered during an expansion. These two items follow the dummy symbol name given in the MACRO directive (as shown in section 5.4 above) if the programmer wishes to use generated symbols in a macro. For example,

```
MUMBLE      MACRO     D,G,4
            < macro body >
            ENDM
```

might contain references to G(1), G(2), G(3), and G(4), these being individual generated symbols.

With regard to generated symbols the macro expansion machinery operates in the following fashion: A generated symbol base value for each macro is initialized to zero at the beginning of assembly. As each generated symbol is encountered, the expression constituting its subscript is evaluated. This value is added to the base value, and the sum is produced as a string of digits concatenated to the generated symbol name; the first digit is always 0 to reduce the likelihood of the generated symbol being identical to a normal symbol defined elsewhere by the programmer. Thus, the first time MUMBLE is called, G(2) will be expanded as GØ2, G(4) as GØ4, etc.

At the end of a macro expansion, the generated symbol base value is incremented by the amount designated by the expression following the generated symbol name in the MACRO directive. This is 4 in the case of MUMBLE. Thus, the second call of MUMBLE will produce in place of G(2), GØ6, the third call will produce GØ1Ø, etc. It should be clear that the generated symbol name should be kept as short as possible.

The expression in the macro head (call it m) must have a value in the range [1,1023]. A generated symbol subscript must have a value in the range [1,m].

5.4.3 Concatenation

Occasionally, it is desirable to have a dummy argument follow immediately after an alphanumeric character, for example, to have D(1) follow just after ALPHA. But then the assembler would not recognize the dummy because it would see ALPHAD(1) instead of D(1). To get around this problem the concatenation symbol '.&' is introduced. Its sole purpose is to separate a dummy argument (or conceivably a generated symbol) from a preceding alphanumeric character during macro definition. Thus, the example becomes ALPHA.&D(1). The concatenation symbol is not stored in string storage so it does not appear during expansion.

As an example, say that we wish to define a macro STORE, and suppose we have established the convention that certain temporary storage cells begin with the letters A, B, or X depending on what register is saved there. The definition is:

```
STORE       MACRO     D
            ST.&D($)    D(1)
            ENDM
```

If called by the statements

```
            STORE     B17
            STORE     X44
```

the macro will expand as

```
            STB       B17
            STX       X44
```

The concatenation symbol may appear anywhere in a macro definition, but it is only necessary in the case described above. If one macro is defined within another, any concatenation symbols within the inner macro will not be removed during the definition of the enclosing macro.

5.4.4 Conversion of a value to a digit string

As an adjunct to the automatic generation of symbols (or for any other purposes for which it may be suited) a capability is provided in the assembler's macro expansion machinery for conversion of the value of an expression at call time to a string of decimal digits. The construct

```
($expression)
```

will be replaced by a string of digits equal to the value of the expression. For example, if X=5 then

```
AB($2*X+1)
```

will be transformed into

```
AB11
```

If the value of the expression is zero then the digit string is '0'; if it is negative then the digit string is preceded by a minus sign.

This conversion scheme can also be used inside repeat blocks; for example

```
          RPT     (I=1,10)
TEMP($I)  BSS     1
          ENDR
```

creates 10 cells labelled TEMP1 through TEMP10.

5.4.5 A note on subscripts

The expressions used as subscripts for dummy arguments and generated symbols, as well as the expressions used in the conversion to a digit string must be absolute. Any undefined symbols appearing in these expressions are treated as if they were defined with the value -1. These expressions may themselves contain dummy arguments, generated symbols, and ($...), so constructs like ($4+D(I*D(3))) are possible.

5.5 NARG and NCHR Number of arguments and number of characters

Macros are more useful if the number of arguments supplied at call time is not fixed. The precise meaning of a macro (and indeed, the result of its expansion) may depend on the number or arrangement of its arguments. In order to permit this, the macro undergoing expansion must be able to determine at call time the number of arguments supplied. The NARG directive makes this possible.

NARG functions like EQU except that no expression is used with it. Its form is

```
        [$]symbol     NARG     [comment]
```

The function of the directive is to equate the value of the symbol to the number of arguments supplied to the macro currently undergoing expansion. The symbol can then be used by itself or in expressions for any purpose. NARG may appear in any macro, even one which has no dummy argument (and thus never has any arguments at call time); it is an error for NARG to appear outside a macro.

It is also useful to be able to determine at call time the number of characters in an argument. NCHR functions by equating the symbol in its label field to the number of characters in its operand field. Its form is

```
        [$]symbol     NCHR     [character string     [comment]]
```

where "character string" has exactly the same form as an argument supplied for a macro call, i.e., if it involves blanks, commas, or semi-colons it should be enclosed in parentheses (see section 5.6). NCHR can appear anywhere, both inside and outside macros, but it is most useful in macros for determining the length of arguments.

Examples:

```
    A    NCHR    ABCDEF            A:=6
    B    NCHR    (,,XYZ,,)         B:=7
    C    NCHR    D(I)              C:=?
```

5.6 Macro calls

The format of a macro call is:

```
[[$]label]      macroname      [argstring]      [comment]
```

Such a call causes the macro whose name appears in the opcode field to be expanded, with the dummy arguments in the macro body replaced by the actual arguments of the argstring.

The label field is always transmitted as argument 0, so that D(el),where el has value 0, is always legal inside a macro. An occurrence of D(el), where el=0, will be replaced by the label field. If the label field is empty, then D(el) expands to the null string. At most seven characters will be transmitted this way: the first six characters of the symbol in the label field, preceded by '$' if the label field begins with '$'.

If the user wishes to transmit an argument to a macro in the label field of the macro call, but does not wish to have the symbol in this field defined, he should define the macro with LMACRO rather than MACRO. (See section 5.4) An example:

```
NT      LMACRO    D
        RPT       D(1)
        DATA      D(2)
        ENDR
D(0)    DATA      -D(1)
        ENDM
```

when called by:

```
DTE     NT        4,4B7
```

expands as:

```
        DATA      4B7
        DATA      4B7
        DATA      4B7
        DATA      4B7
DTE     DATA      -4
```

Notice that this would have caused a doubly-defined symbol error had MACRO been used rather than LMACRO.

A macro call may or may not have an arg string (see section 5.4). If an arg string is present, it may contain any number of arguments, in fact, more than are referred to by the macro.

Before describing an arg string, the following should be noted: blanks, commas, semi-colons, and parentheses that are enclosed in single or double quotes are treated exactly like ordinary characters enclosed in quotes; they do not serve as terminators, separators, delimiters, or the like. In effect, when the argument collector in NARP is collecting arguments for a macro call, the occurrence of a quote causes it to stop looking for special characters except for a matching quote (and, of course, carriage return, which is an absolute terminator). A single quote enclosed in double quotes is not a special character and vice versa. Thus, when a blank, comma, semi-colon, or parenthesis is referred to in the following, it is understood that it is not enclosed in quotes.

An arg string for a macro call has the following format:

<arg>,<arg>,...,<arg> <terminator>

where a terminator is a blank, semi-colon, or carriage return. There are three forms of <arg>:

1. <arg> may be the null string.
2. If the first character of <arg> is not a left parenthesis then <arg> is a string of characters not containing blank, comma, semi-colon, or carriage return (remember that blanks, commas, and semi-colons may appear in <arg> if they are enclosed in quotes).
3. If the first character of <arg> is a left parenthesis the <arg> does not terminate until a blank, comma, or semi-colon is encountered after the right parenthesis which matches the initial left parenthesis ("matches" means that all left and right parentheses in the argument are noted and paired off with each other so that a nested parentheses structure is possible).
   Of course, a carriage return at any point immediately

terminates <arg>. Again, remember that blanks, commas, semi-colons, and parentheses enclosed in quotes are ignored when <arg> is being delimited. The initial left parenthesis and its matching right parenthesis (which need not be the last character in <arg>) are removed before <arg> is transmitted to the macro.
Examples:

```
AMAC        (,␣;␣,),,'HOUSE,␣ROGER',(AB")")
   D(1) =   ,␣;␣,
   D(2) =   null string
   D(3) =   'HOUSE,␣ROGER'
   D(4) =   AB")"
```

5.7 Examples of conditional assembly and macros

1. It is desired to have a pair of macros SAVE and RESTOR for saving and restoring active registers at the beginning and end of subroutines. These macros should take a variable number of arguments so that, for example, one can write

```
SAVE        A,SUBRS
RESTOR      A,B,X,SUBRS
```

to generate the code

```
STA         SUBRSA
LDA         SUBRSA
LDB         SUBRSB
LDX         SUBRSX
```

To this end we first define a macro MOVE which is called by the same arguments delivered to SAVE and RESTOR, but with the string 'ST' or 'LD' appended.

```
MOVE        MACRO       D
X           NARG
            RPT         (Y=2,X-1)
            D(1)D(Y)    D(X)D(Y)
            ENDR
            ENDM
```

Now SAVE and RESTOR can be defined as

```
SAVE        MACRO       D
            MOVE        ST,D(,)
            ENDM
```

```
RESTOR    MACRO     D
          MOVE      LD,D(,)
          ENDM
```

2. Many programmers use flags, memory cells that are used as binary indicators. The instruction SKN (skip if memory negative) makes it easy to test these flags if the convention is used that a flag is set (true) if it contains -1 and reset (false) if it contains Ø. We want to define two macros, SET and RESET to manipulate these flags; furthermore, it is desirable to deliver at call time the name of an active register which will be used for the action. Calls of the macros will look like

```
SET       A,FLG1,FLG2,FLG3
RESET     X,FLG37,FLG12
```

As in the previous example we make use of an intermediate macro, STORE, which takes the same arguments as SET and RESET.

```
STORE     MACRO     D
X         NARG
          RPT       (Y=2,X)
          ST.&D(1)  D(Y)
          ENDR
          ENDM
```

Now SET and RESET are defined as

```
SET       MACRO     D
          LD.&D(1)  =-1
          STORE     D(,)
          ENDM

RESET     MACRO     D
          CL.&D(1)
          STORE     D(,)
          ENDM
```

3. The following macro, MOVE, takes any number of pairs of arguments; the first argument of each pair is moved to the second, but an argument may itself be a pair of arguments, which may themselves be pairs of arguments, etc. MOVE extracts pairs of argument structures and transmits them to a second macro MOVE1.

```
MOVE      MACRO     D
X         NARG
          RPT       (Y=1,2,X)
          MOVE1     D(Y),D(Y+1)
          ENDR
          ENDM
```

The main work is done in MOVE1 which calls itself recursively until it comes up with a single pair of arguments.

```
MOVE1   MACRO   D,G,2
G(1)    NARG
G(2)    EQU     Ø
        IF      G(1)=2
        LDA     D(1)
        STA     D(2)
        ELSE
        RPT     G(1)/2,(G(2)=G(2)+1)
        MOVE1   D(G(2)),D(G(2)+G(1)/2)
        ENDR
        ENDF
        ENDM
```

When MOVE is called by

```
        MOVE    A,B
```

the code generated is

```
        LDA     A
        STA     B
```

When called by

```
        MOVE    A,B,C,D
```

the code generated is

```
        LDA     A
        STA     B
        LDA     C
        STA     D
```

When called by

```
        MOVE    (A,B),(C,D)
```

the code generated is

```
        LDA     A
        STA     C
        LDA     B
        STA     D
```

And when called by

```
MOVE      ((A,B),(C,D)),((E,F),(G,H))
```

the code generated is

```
LDA       A
STA       E
LDA       B
STA       F
LDA       C
STA       G
LDA       D
STA       H
```

It is instructive to trace the last example by hand to see how the recursive calls of MOVE1 work. This is an exercise left to the reader.

6.0 Operating NARP

6.1 Error comments on statements

When NARP encounters a statement which it deems incomprehensible or illegal, it lists the statement in error-format (corresponds to all listing format booleans being set) and then on the following line(s) lists all error comments pertaining to the statement.

Most error-comments are as intelligible as the situation (and NARP's strangeness) allows. Some of the more common and/or more obscure ones are listed and commented upon below:

C ?	The character C caught NARP unawares
BAD TERMINATION	Premature termination, or garbage (like extraneous commas) where the statement should end.
LC OVERFLOW	The value of the location counter got out of the range [0, 37777B].
DIRECTIVE OUTSIDE BODY	And ENDF, ENDR, or ENDM without a matching IF, RPT, or MACRO.
(symb) REDEFINED	"symb" was defined (as a label) previous to this definition of it.
(symb) OPCODE?	"symb" was used as an opcode and is not in the opcode table.
UNDEFINED EXPRESSION	An undefined symbol occurs in an expression which should be defined.

6.2 Other error comments

If a fixed-length table ever flows, a message (name) OVERFLOW is printed (after a listing of the offending statement in error-format), followed by *****ASSEMBLY DEAD***** and termination of one assembly.

The name may be:

MAIN TABLE	Contains opcodes, literals, symbols (both undefined and defined).
STRING STORAGE	Contains MACRO definitions, macro calls and RPT expresions.

EXPRESSION TABLE	Contains post-fix Polish representations of expressions containing undefined symbols, until all the symbols in the expression are defined.
INPUT POINTER STACK	Contains one entry for each embedded change of input-source.
CHARACTER STACK	Holds the characters in a symbol while they are being collected.
OPERAND STACK	Holds operands in the processing of expressions.
PILE	Space for temporaries in recursive calls of the expression eater.

In addition, the following comments may appear:

TRAP AT XXXXX	Error committed by NARP at location XXXXX; assembly terminates.
I-O ERROR	Error in input or output of information, assembly terminates.
NO END DIRECTIVE	An end-of-file encountered before an END directive; assembly will terminate as though an END directive was given (i.e., normally).

6.3 Starting an assembly

Assuming that the user has entered the time-sharing system, NARP is called by hitting the rubout button until the exec answers (by typing '@') and then typing 'NARP' followed by a dot. Control is then turned over to NARP and a source file must be specified; other information may also be supplied, if desired. The general format is:

		default convention
@NARP.		
SOURCE FILE:	file name	none
OBJECT FILE:	file name	none
[TEXT FILE:	file name]	TELETYPE

Each line above is either terminated by a dot or a semi-colon. A dot causes assembly to begin immediately (except after the source file name). The default conventions are used for all those options not explicitly specified. A semi-colon causes a carriage return to be typed, and the specification of some option is expected.

The various options are discussed in more detail below:

SOURCE FILE: As soon as NARP is started this line is typed and the user must specify a file containing a program to be assembled. When he terminates the name, NARP responds with 'OBJECT FILE:' on the next line.

OBJECT FILE: The file name given specifies where the binary output from the program should go. If the file name is terminated by a semi-colon, then a carriage return is typed and NARP waits for one of the following options to be specified.

TEXT FILE: The file name given specifies where the listing of the source program and of the error messages should go. This option may be specified only once.

Appendix A: List of all pre-defined opcodes and pre-defined symbols

The following table is a listing of an initialization program used to initialize the opcode table and symbol table of NARP. It will be noted that in some cases the OPD directive has four operands instead of the usual three; the fourth operand specifies the type (directive, macro, or instruction) of the opcode being defined. It is only possible to use four operands for OPD when NARP is being initialized, and once the initialization program has been assembled, OPD will only accept three operands.

```
* NARP INITIALIZATION PROGRAM.                          (21 NOV 1966)

* OPD SYNTAX AND SEMANTICS:
* <SYMBOL> OPD   <VALUE>[,<OP SIT>[,<SHIFTK>[,<TYPE>]]]

* <OPSIT>    :      Ø - OPERAND OPTIONAL
*                   1 - NO OPERAND
*                   2 - OPERAND REQUIRED

* <SHIFTK>   :      Ø - NORMAL INSTRUCTION
*                   1 - SHIFT INSTRUCTION

* <TYPE>     :      Ø - INSTRUCTION
*                   1 - DIRECTIVE
*                   2 - MACRO

* INSTRUCTION DEFINITIONS:

LDA     OPD     Ø76ØØØØØB,2     LOAD A
STA     OPD     Ø35ØØØØØB,2     STORE A
LDB     OPD     Ø75ØØØØØB,2     LOAD B
STB     OPD     Ø36ØØØØØB,2     STORE B
LDX     OPD     Ø71ØØØØØB,2     LOAD X
STX     OPD     Ø37ØØØØØB,2     STORE X
EAX     OPD     Ø77ØØØØØB,2     COPY EFFECTIVE ADDRESS INTO X
XMA     OPD     Ø62ØØØØØB,2     EXCHANGE M AND A

ADD     OPD     Ø55ØØØØØB,2     ADD M TO A
ADC     OPD     Ø57ØØØØØB,2     ADD WITH CARRY
ADM     OPD     Ø63ØØØØØB,2     ADD A TO M
MIN     OPD     Ø61ØØØØØB,2     MEMORY INCREMENT
SUB     OPD     Ø54ØØØØØB,2     SUBTRACT M FROM A
SUC     OPD     Ø56ØØØØØB,2     SUBTRACT WITH CARRY
MUL     OPD     Ø64ØØØØØB,2     MULTIPLY
DIV     OPD     Ø65ØØØØØB,2     DIVIDE

ETR     OPD     Ø14ØØØØØB,2     EXTRACT (AND)
MRG     OPD     Ø16ØØØØØB,2     MERGE (OR)
EOR     OPD     Ø17ØØØØØB,2     EXCLUSIVE OR

RCH     OPD     Ø46ØØØØØB,2     REGISTER CHANGE
CLA     OPD     Ø46ØØØØ1B,1     CLEAR A
CLB     OPD     Ø46ØØØØ2B,1     CLEAR B
CLAB    OPD     Ø46ØØØØ3B,1     CLEAR AB
CLX     OPD     246ØØØØØB,1     CLEAR X
CLEAR   OPD     246ØØØØ3B,1     CLEAR A, B, AND X
CAB     OPD     Ø46ØØØØ4B,1     COPY A INTO B
CBA     OPD     Ø46ØØØ1ØB,1     COPY B INTO A
XAB     OPD     Ø46ØØØ14B,1     EXCHANGE A AND B
BAC     OPD     Ø46ØØØ12B,1     COPY B INTO A, CLEARING B
ABC     OPD     Ø46ØØØØ5B,1     COPY A INTO B, CLEARING A
CXA     OPD     Ø46ØØ2ØØB,1     COPY X INTO A
```

```
CAX      OPD      04600400B,1       COPY A INTO X
XXA      OPD      04600600B,1       EXCHANGE X AND A
CBX      OPD      04600020B,1       COPY B INTO X
CXB      OPD      04600040B,1       COPY X INTO B
XXB      OPD      04600060B,1       EXCHANGE X AND B
STE      OPD      04600122B,1       STORE EXPONENT
LDE      OPD      04600140B,1       LOAD EXPONENT
XEE      OPD      04600160B,1       EXCHANGE EXPONENTS
CNA      OPD      04601000B,1       COPY NEGATIVE OF A INTO A
AXC      OPD      04600401B,1       COPY A TO X, CLEAR A

BRU      OPD      00100000B,2       BRANCH UNCONDITIONALLY
BRX      OPD      04100000B,2       INCREMENT INDEX AND BRANCH
BRM      OPD      04300000B,2       MARK PLACE AND BRANCH
BRR      OPD      05100000B,2       RETURN BRANCH
BRI      OPD      01100000B,2       BRANCH AND RETURN FROM INTERRUPT

SKS      OPD      04000000B,2       SKIP IF SIGNAL NOT SET
SKE      OPD      05000000B,2       SKIP IF A EQUALS M
SKG      OPD      07300000B,2       SKIP IF A GREATER THAN M
SKR      OPD      06000000B,2       REDUCE M, SKIP IF NEGATIVE
SKM      OPD      07000000B,2       SKIP IF A EQUALS M ON B MASK
SKN      OPD      05300000B,2       SKIP IF M NEGATIVE
SKA      OPD      07200000B,2       SKIP IF M AND A DO NOT COMPARE ONES
SKB      OPD      05200000B,2       SKIP IF M AND B DO NOT COMPARE ONES
SKD      OPD      07400000B,2       DIFFERENCE EXPONENTS AND SKIP

RSH      OPD      06600000B,2,1     RIGHT SHIFT AB
RCY      OPD      06620000B,2,1     RIGHT CYCLE AB
LRSH     OPD      06624000B,2,1     LOGICAL RIGHT SHIFT AB
LSH      OPD      06700000B,2,1     LEFT SHIFT AB
LCY      OPD      06720000B,2,1     LEFT CYCLE AB
NOD      OPD      06710000B,2,1     NORMALIZE AND DECREMENT X

HLT      OPD      00000000B,0       HALT
ZRO      OPD      00000000B,0       ZERO
NOP      OPD      02000000B,0       NO OPERATION
EXU      OPD      02300000B,2       EXECUTE

BPT1     OPD      04020400B,1       BREAKPOINT TEST 1
BPT2     OPD      04020200B,1       BREAKPOINT TEST 2
BPT3     OPD      04020100B,1       BREAKPOINT TEST 3
BPT4     OPD      04020040B,1       BREAKPOINT TEST 4

ROV      OPD      02200001B,1       RESET OVERFLOW
REO      OPD      02200010B,1       RECORD EXPONENT OVERFLOW
OVT      OPD      02200101B,1       OVERFLOW TEST AND RESET
OTO      OPD      02200100B,1       OVERFLOW TEST ONLY

EIR      OPD      00220002B,1       ENABLE INTERRUPTS
DIR      OPD      00220004B,1       DISABLE INTERRUPTS
AIR      OPD      00220020B,1       ARM/DISARM INTERRUPTS
IET      OPD      04020002B,1       INTERRUPT ENABLED TEST
IDT      OPD      04020004B,1       INTERRUPT DISABLED TEST
```

```
ALCW     OPD      00250000B,1      ALERT CHANNEL W
DISW     OPD      00200000B,1      DISCONNECT CHANNEL W
ASCW     OPD      00212000B,1      ALERT TO STORE ADDRESS IN CHANNEL W
TOPW     OPD      00214000B,1      TERMINATE OUTPUT ON CHANNEL W

CATW     OPD      04014000B,1      CHANNEL ACTIVE TEST
CETW     OPD      04011000B,1      CHANNEL W ERROR TEST
CZTW     OPD      04012000B,1      CHANNEL W COUNT TEST
CITW     OPD      04010000B,1      CHANNEL W INTER-RECORD TEST

EOD      OPD      00600000B,2      ENERGIZE OUTPUT D
MIW      OPD      01200000B,2      M INTO W BUFFER WHEN EMPTY
WIM      OPD      03200000B,2      W BUFFER INTO M WHEN FULL
PIN      OPD      03300000B,2      PARALLEL INPUT
POT      OPD      01300000B,2      PARALLEL OUTPUT
EOM      OPD      00200000B,2      ENERGIZE OUTPUT M
BETW     OPD      04020010B,1      W BUFFER ERROR TEST
BRTW     OPD      04021000B,1      W BUFFER READY TEST

BIO      OPD      57600000B,2      BLOCK I/O
BRS      OPD      57300000B,2      BRANCH TO SYSTEM
CIO      OPD      56100000B,2      CHARACTER I/O
CTRL     OPD      57200000B,2      CONTROL
DBI      OPD      54200000B,2      DRUM BLOCK INPUT
DBO      OPD      54300000B,2      DRUM BLOCK OUTPUT
DWI      OPD      54400000B,2      DRUM WORD INPUT
DWO      OPD      54500000B,2      DRUM WORD OUTPUT
EXS      OPD      55200000B,2      EXECUTE INSTRUCTION IN SYSTEM MODE
FAD      OPD      55600000B,2      FLOATING ADD
FDV      OPD      55300000B,2      FLOATING DIVIDE
FMP      OPD      55400000B,2      FLOATING MULTIPLY
FSB      OPD      55500000B,2      FLOATING SUBTRACT
GCD      OPD      53700000B,2      GET CHARACTER AND DECREMENT
GCI      OPD      56500000B,2      GET CHARACTER AND INCREMENT
ISC      OPD      54100000B,2      INTERNAL TO STRING CONV. (FLOATING OUTPUT)
IST      OPD      55000000B,2      INPUT FROM SPECIFIED TELETYPE
LAS      OPD      54600000B,2      LOAD FROM SECONDARY MEMORY
LDP      OPD      56600000B,2      LOAD POINTER (AB)
OST      OPD      55100000B,2      OUTPUT TO SPECIFIED TELETYPE
SAS      OPD      54700000B,2      STORE IN SECONDARY MEMORY
SBRM     OPD      57000000B,2      SYSTEM BRM
SBRR     OPD      05140000B,2      SYSTEM BRR
SIC      OPD      54000000B,2      STRING TO INTERNAL CONV. (FLOATING INPUT)
SKSE     OPD      56300000B,2      SKIP IF STRINGS EQUAL
SKSG     OPD      56200000B,2      SKIP IF STRING GREATER
STI      OPD      53600000B,2      SIMULATE TELETYPE INPUT
STO      OPD      53400000B,2      STEAL TTY OUTPUT
STP      OPD      56700000B,2      STORE POINTER (AB)
TCI      OPD      57400000B,2      TELETYPE CHARACTER INPUT
TCO      OPD      57500000B,2      TELETYPE CHARACTER OUTPUT
WCD      OPD      53500000B,2      WRITE CHARACTER AND DECREMENT
WCH      OPD      56400000B,2      WRITE CHARACTER
WCI      OPD      55700000B,2      WRITE CHARACTER AND INCREMENT
```

```
[illegible]0    OPD     56ØØØØØØB,2     WORD I/O


* DIRECTIVE DEFINITIONS:

ASC     OPD     Ø,2,Ø,1         ASCII STRING
BES     OPD     1,2,Ø,1         BLOCK END SYMBOL
BSS     OPD     2,2,Ø,1         BLOCK START SYMBOL
COPY    OPD     3,2,Ø,1         REGISTER CHANGE
CRPT    OPD     4,2,Ø,1         CONDITIONAL REPEAT
DATA    OPD     5,2,Ø,1         DATA WORD
DEC     OPD     7,1,Ø,1         SET NUMBER RADIX TO 1Ø
DELSYM  OPD     8,1,Ø,1         DELETE SYMBOL
ELSE    OPD     9,1,Ø,1         ELSE
ELSF    OPD     1Ø,2,Ø,1        ELSE IF
END     OPD     11,1,Ø,1        END OF PROGRAM
ENDF    OPD     12,1,Ø,1        END IF
ENDM    OPD     13,1,Ø,1        END MACRO
ENDR    OPD     14,1,Ø,1        END REPEAT
EQU     OPD     15,2,Ø,1        EQUATE
EXT     OPD     16,Ø,Ø,1        EXTERNAL
FREEZE  OPD     17,1,Ø,1        FREEZE TABLES
FRGT    OPD     18,2,Ø,1        FORGET SYMBOL
IDENT   OPD     19,1,Ø,1        IDENTIFICATION SYMBOL
IF      OPD     2Ø,2,Ø,1        IF
LIBEXT  OPD     25,1,Ø,1        OUTPUT LABEL AS LIBRARY WYMBOL
LIST    OPD     21,Ø,Ø,1        TURN ON LISTING
MACRO   OPD     22,Ø,Ø,1        MACRO DEFINITION
NARG    OPD     23,1,Ø,1        NUMBER OF ARGUMENTS
NCHR    OPD     24,Ø,Ø,1        NUMBER OF CHARACTERS
NOLIST  OPD     26,Ø,Ø,1        TURN OFF LISTING
OCT     OPD     27,1,Ø,1        SET NUMBER RADIX TO 8
POPD    OPD     28,2,Ø,1        POP DEFINITION
RELORG  OPD     29,2,Ø,1        RELATIVE ORIGIN
RETREL  OPD     3Ø,1,Ø,1        RETRIEVE ORIGIN
RPT     OPD     31,2,Ø,1        REPEAT
TEXT    OPD     32,2,Ø,1        STRING (FOUR CHARACTERS PER WORD)
LMACRO  OPD     33,Ø,Ø,1        ALTERNATIVE MACRO DEF'N
REM     OPD     35,2,Ø,1        PRINT REMARK ON TEXT FILE
FRGTOP  OPD     37,2,Ø,1        FORGET SELECTED OPCODES

:ZERO:  EQU     *
:LC:    EQU     :ZERO:
        FRGT    :ZERO:,:LC:


        FREEZE

        END                     LAST LINE OF NARP INITIALIZATION PROGRAM.
```

Appendix B: Table of ASCII character set for the SDS 940

octal value	character	octal value	character	octal value	character
0		30	8	60	P
1	!	31	9	61	Q
2	"	32	:	62	R
3	#	33	;	63	S
4	$	34	<	64	T
5	%	35	=	65	U
6	&	36	>	66	V
7	'	37	?	67	W
10	(	40	@	70	X
11	)	41	A	71	Y
12	*	42	B	72	Z
13	+	43	C	73	[
14	,	44	D	74	\
15	-	45	E	75	]
16	.	46	F	76	↑
17	/	47	G	77	←
20	Ø	50	H	135	MULTIPLE BLANKS
21	1	51	I	137	END-OF-FILE
22	2	52	J	144	END-OF-TAPE
23	3	53	K	147	BELL
24	4	54	L	152	LF
25	5	55	M	154	START NEW PAGE
26	6	56	N	155	CR
27	7	57	O		