ARPAS

REFERENCE MANUAL FOR TIME SHARING

ASSEMBLER FOR SDS 930

Document No. R-26

Revised February 24, 1967

Contract SD-185

Office of Secretary of Defense

Advanced Research Projects Agency

Washington 25, D.C.

TABLE OF CONTENTS

APPENDIX A

EXTENDED LIST OF INSTRUCTIONS

Mnemonic	Operation Code	Function
Load/Store		
LDA	76	Load A
STA	35	Store A
LDB	75	Load B
STB	36	Store B
LDX	71	Load X
STX	37	Store index
EAX	77	Copy effective address into index
XMA	62	Exchange M and A
Arithmetic		
ADD	55	Add M to A
ADC	57	Add with carry
ADM	63	Add A to M
MIN	61	Memory increment
SUB	54	Subtract M from A
SUC	56	Subtract with carry
MUL	64	Multiply
DIV	65	Divide
Logical		
ETR	14	Extract (AND)
MRG	16	Merge (OR)
EOR	17	Exclusive or
Register Change		
RCH	46	Register change
CLA	0 46 00001	Clear A
CLB	0 46 00002	Clear B
CLAB	0 46 00003	Clear AB
CLX	2 46 00000	Clear X
CLEAR	2 46 00003	Clear A, B and X
CAB	0 46 00004	Copy A into B

Mnemonic	Operation Code	Function
CBA	0 46 00010	Copy B into A
XAB	0 46 00014	Exchange A into B
BAC	0 46 00012	Copy B into A, Clearing B
ABC	0 46 00005	Copy A into B, Clearing A
CXA	0 46 00200	Copy X into A
CAX	0 46 00400	Copy A into X
XXA	0 46 00600	Exchange X and A
CBX	0 46 00020	Copy B into X
CXB	0 46 00040	Copy X into B
XXB	0 46 00060	Exchange X and B
STE	0 46 00122	Store Exponent
LDE	0 46 00140	Load Exponent
XEE	0 46 00160	Exchange Exponents
CNA	0 46 01000	Copy negative into A
AXC	0 46 00401	Copy A to X, clear A
Branch		
BRU	01	Branch unconditionally
BRX	41	Increment index and branch
BRM	43	Mark place and branch
BRR	51	Return branch
BRI	11	Branch and return from interrupt
Test/Skip		
SKS	40	Skip if signal not set
SKE	50	Skip if A equals M
SKG	73	Skip if A greater than M
SKR	60	Reduce M, skip if negative
SKM	70	Skip if A = M on B mask
SKN	53	Skip if M negative
SKA	72	Skip if M and A do not compare ones
SKB	52	Skip if M and B do not compare ones
SKD	74	Difference exponents and skip

Mnemonic	Operation Code	Function
Shift		
RSH	0 66 00xxx	Right shift AB
RCY	0 66 20xxx	Right cycle AB
LRSH	0 66 24xxx	Logical right shift
LSH	0 67 00xxx	Left shift AB
LCY	0 67 20xxx	Left cycle AB
NOD	0 67 10xxx	Normalize and decrement X
Control		
HLT, ZRO	00	Halt
NOP	20	No operation
EXU	23	Execute
Breakpoint Tests		
BPTx	0 40 20xx0	Breakpoint test
Overflow		
ROV	0 22 00001	Reset overflow
REO	0 22 00010	Record exponent overflow
OVT	0 22 00101	Overflow test and reset
OTO	0 22 00100	Overflow test only
Interrupt		
EIR	0 02 20002	Enable interrupts
DIR	0 02 20004	Disable interrupts
AIR	0 02 20020	Arm/disarm interrupts
IET	0 40 20002	Interrupt enabled test
IDT	0 40 20004	Interrupt disabled test
Channel Tests		
CATW	0 40 14000	Channel W active test
CETW	0 40 11000	Channel W error test
CZTW	0 40 12000	Channel W zero count test
CITW	0 40 10000	Channel W inter-record test
Input/Output		
EOD	06	Energize output D

Mnemonic	Operation Code	Function
Input/Output (920 Compatible)		
MIW	12	M into W buffer when empty
WIM	32	W buffer into M when full
PIN	33	Parallel input
POT	13	Parallel output
EOM	02	Energize output M
BETW	0 40 20010	W buffer error test
BRTW	0 40 21000	W buffer ready test
Syspops		
BIO	576	Block I/O
BRS	573	Branch to system
CIO	561	Character I/O
CTRL	572	Control
DBI	542	Drum block input
DBO	543	Drum block output
DWI	544	Drum word input
DWO	545	Drum word output
EXS	552	Execute instruction in system mode
FAD	556	Floating add
FDV	553	Floating divide
FMP	554	Floating multiply
FSB	555	Floating subtract
GCD	537	Get character and decrement
GCI	565	Get character and increment
ISC	541	Internal to string conversion (floating output)
IST	550	Input from specified teletype
LAS	546	Load from secondary memory
LDP	566	Load pointer (AB)
LIO	552	Link I/O
OST	551	Output to specified teletype
SAS	547	Store in secondary memory
SBRM	570	System BRM
SBRR	51*	System BRR (prestored macro)
SIC	540	String to internal conversion (floating input)
SKSE	563	Skip on string equal
SKSG	562	Skip on string greater

Mnemonic	Operation Code	Function
STI	536	Simulate teletype input
STP	567	Store pointer
TCI	574	Teletype character input
TCO	575	Teletype character output
WCD	535	Write character and decrement
WCH	564	Write character
WCI	557	Write character and increment
WIO	560	Word I/O

1.0 Introduction

An assembler is a translator whose source language is assembly language and whose object code is actual machine language. Assembly language is mostly a one-for-one representation of machine language written in a symbolic form. Its value comes from being easier to read and from the facilities provided by the assembler for doing calculations at assembly time. These range from simple address calculations to complex conditional assemblies in which totally different object programs may be generated, with the choice among them depending on the values of a few parameters.

This section serves to define the terminology used. It is assumed that the programmer is familiar with the basic characteristics of the SDS 940*.

1.1 Basic Description of the Assembler

The assembler is a two-pass assembler with subprogram, literal, macro, and conditional assembly capabilities.

1.2 Symbols

Numbers may be represented symbolically in assembly language by symbols. A symbol is any string of letters and digits not forming a constant. (Constants are defined in Section 4.2). In particular, it is not necessary that a symbol begin with a letter. Although symbols as written may be arbitrarily long, only the first six characters of a symbol are used to distinguish it from others. When a symbol is used to represent a memory address, it is called a label. Examples of symbols are:

```
START   Z1C   A12   CALCULATE
```

* Ref. to SDS 940 Computer Reference Manual, No. 90 06 40A, August, 1966.

1.3 Instructions, Directives, and Comments

Input to the assembler takes the form of a sequence of statements called instructions, directives, or comments. Instructions are symbolic representations of machine commands and are translated by the assembler into machine language. Directives, by contrast, are messages which serve to control the assembly process or create data. They may or may not generate output. Comments are ignored by the assembler, and serve only to clarify the meaning of a program.

1.4 Subprograms

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.

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.

1.5 Literals

Often data is placed in programs at assembly time. It is frequently convenient to refer to constants by value than by label. A literal is a symbolic reference to a datum by value. The assembler allows any type of expression to be used as a literal. Some examples of literals are:

```
=5      =3*XYZ-2    ='END'     =EXTERN
```

1.6 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 to the loader as a binary quantity called the relocation value. If R = 1 the operand is to be relocated; if R = 0 the operand is absolute.

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.

It is possible to disable relocation in the assembler and to do absolute assembly. In this event there is an option which produces a paper tape which can be loaded using the 940 fill switch.

1.7 Basic Assembly Procedure

During pass 1 of the two-pass process the operands of instructions and some directives are scanned for the presence of single symbols. If a single symbol is present, a table of symbols is searched. If absent, the symbol is added to the table but marked as not yet defined, i.e., having no value. Labels are placed into the symbol table in similar fashion, except that they are assigned the current value of the location counter, a word within the assembler which contains the relative address of the instruction. If a label has been previously defined, it is marked as a duplicate symbol (this is taken to be an error).

At the end of pass 1 the symbol table is sorted. All symbols present having no value are assumed to be external. These symbols are then output by the assembler for later use by the loader. During pass 2 the labels are not computed; rather, the operand fields of instructions and directives are evaluated using the now known symbol values.

In absolute assemblies the scan for single symbols in pass 1 is disabled. This has the effect of doing away with external symbols.

1.8 Notation

In the following pages, square brackets [ ] are used to indicate the presence of optional quantities.

2.0 The Assembly Language

2.1 Character Set

The classes of characters recognized by the assembler are as follows:

(a) digits

(1) octal 0-7

(2) decimal 0-9

(b) letters A-Z

(c) alphanumerics 0-9 and A-Z

(d) delimiters + - * / , ' ( ) = . $ blank ←

(e) special characters : ; < > ? [ ] "

Note that the characters ! # % & @ \ ↑ which are normally found on standard Teletypes are not recognized by the assembler. Use of them in a program will result in their being replaced by blanks.

2.2 Statements

Statements are logical units of input. They may be delimited either by being placed on separate lines or by being separated with semi-colons. Semi-colons do not serve as statement delimiters when used between single quotes (as in the TEXT directive) or inside of matched parentheses (as in arguments of macro calls). Examples of statements are

```
START   LDA     DAT21
        MUL     21B
        STA     ANSWER
```

or

```
START   LDA   DAT21;  MUL  21B;  STA  ANSWER
```

If a statement requires more than one line for any reason, it can be continued on the next line by typing a + in the first column of the next lin

Thus:

```
START  LDA  DAT21;  MUL  21B;  STA  ANSWER   THE COM
+MENT ON THIS LINE REQUIRES A CONTINUATION
```

This kind of continuation may be done for about five lines (320 characters).

Each non-blank statement is an instruction, a directive, or a comment. Blank statements are ignored. Comments begin with an asterisk; they have absolutely no effect on the program being assembled and serve only as annotations to clarify the meaning of the assembly language.

Directives and instructions are divided into four fields. The fields are, from left to right, the label field, the operation field, the operand field, and the comment field. The assembler is a free-form assembler; its various fields are delimited by blanks rather than restricting them to fixed places in a line. This is explained in more detail below.

The label field is used mostly for symbol definitions. It begins with the first character in the statement and ends on the first non-alphanumeric character. (The blank is usually the only legal terminator.) Thus, in the following statements the symbol XYZ appears in label fields.

```
XYZ    LDA =10
 STA DEF;XYZ  LDA =10;    LDB* LMN
```

The operation field contains (usually) a symbolic operation code or directive name. It begins with the first non-blank character after the termination of the label field. In the statements above, each operation field begins in a different position. Like the label field, the operation field terminates on the first non-alphanumeric character. Legal terminators are the blank, asterisk, semi-colon, and carriage return.

The operand and comment fields each begin with the first non-blank character after the termination of the preceding field. The operand field terminates on the first blank or semi-colon not between matched single quotes or parentheses. The carriage return always terminates the field (and the statement). The comment field terminates on a semi-colon or carriage return. This field, like the comment statement, is not used by the assembler; it may contain anything.

2.3 Programs

A program consists of a sequence of statements terminated by an END directive. Normally programs are assembled in relocatable form. A program is assembled in absolute self-loading form if it begins with an ORG directive. It is possible (by using RELORG) to make an absolute assembly to be loaded by DDT.

3.0 The Syntax of Instructions

3.1 Their Classification

(a) Class 1 (normal instructions).

Class 1 instructions in general use the operand field. Its absence implies the value zero. It is possible to specify for each Class 1 instruction whether or not the operand field must be present. It is also possible to specify that bit 0 of the instruction word is to be set to one (as in SYSPOPs). There are two types of Class 1 instructions:

(1) type 0

The address is formed mod 2^{14}. All instructions making memory references are of this type.

(2) type 1

The operand is formed mod 2^{9}. This type is used for shift instructions. If indirect addressing is used with this type, the address is formed mod 2^{14}.

Class 1 instructions have the following form:

[[$]label] opcode[*] [operand[,tag]] [comment]

Indirect addressing is signified by an asterisk immediately following the operation code or by preceding the operand with ← .

The use of the dollar sign is explained in 3.2 The tag is used to specify bits 0, 1 and 2 of the 940 instruction word.

(b) Class 2 (complete or full word instructions).

Class 2 instructions have no operand field. Indirect addressing is signified by an asterisk immediately following the operation code. Class 2 instructions have the following form:

[[$]label] opcode[*] [comment]

(c) Numeric op codes.

Operation codes may be specified as decimal or octal numbers, as for example:

```
[[$]label]  76B[*] [operand[,tag]] [comment]
```

The assembler shifts the numeric op code (modulo 177_8) left to the correct position in the instruction word. In such cases, the op code is assumed to be Class 1, type 0, no operand required, and with bit 0 not set.

3.2 Use of the Label Field

A label identifies the instruction or data word being generated. The symbol used in the label field is given the current value of the location counter. Instructions will have labels normally if they are referred to elsewhere in the program, although it is not necessary that symbols defined in this way be used in references. Symbols defined but not used are called nulls; they are marked as such in the assembly listing and explicitly typed out at the end of an assembly.

If the same symbol appears in the label field of more than one instruction, it is marked as a duplicate and given the newer value.

A $ preceding a label causes an external symbol definition (cf. 6.6).

3.3 Operand Field

The operand field contains at most two arithmetic expressions (or a literal and one expression) used to determine the operand and tag of the machine command. The tag, if present, is evaluated mod 2^3 and must be absolute (i.e. non-relocatable).

3.4 Alternate Conventions for Expressing Indexed & Indirect Addresses

It is possible to express both the use of indexing and indirect addressing in an alternative manner. In each case a special character is placed at the beginning of the operand field. These characters are / for indexing and ← for indirect addressing. Thus, for example,

LDA VECTOR,2 is the same as LDA /VECTOR

and

STA* POINTR is the same as STA ←POINTR

Similarly,

LDA* COMPLX,2 may be written either as

LDA /←COMPLX

or LDA ←/COMPLX

Anything normally useful may follow the initial ← or /, for example

LDA←=CHAIN (LDA* =CHAIN)

This alternate way of expressing indexing and indirect addressing may be used by programmers as they choose. It was devised to simplify the indication of these operations in the use of macros (see chapter 7).

3.5 Comment Field

The comment field is not processed by the assembler, but is copied to the assembly listing.

4.0 Expression Syntax

The assembler evaluates expressions as 24-bit, signed integers. Expressions consist of constants and symbols connected by operators. Examples of expressions are:

```
100-2*ABC(OR)DEF/27B
22
C12>D19
```

Expressions are evaluated from left to right, some operators taking precedence over others. As an expression is evaluated, a parallel calculation of its relocation value R is made. Only absolute expressions (R = 0) and relocatable expressions (R = 1) are legal (cf. 4.7).

4.1 Operators

The operators recognized by the assembler and their precedence are given below. Operators of highest precedence are applied first in evaluation of expressions.

	Operator	Precedence
(a) unary		
	+	4
	-	4
	(NOT)	4
	(R)	4 (cf. 4.7)
(b) relational		
	(LSS) or <	3
	(GRT) or >	3
	(EQU) or =	3
(c) binary		
	*	2
	/	2
	(AND)	2
	+	1
	-	1
	(OR)	1
	(EOR)	1

Note that some operators are more than one character long. These are enclosed in parentheses to avoid confusion with symbols which would otherwise look the same. Parentheses are therefore not allowed in expressions to delineate terms and modify the order of evaluation.

The relational operators give rise to a value 1 if the relation is true and 0 if false. There may be only one relational operator in an expression.

4.2 Constants

Constants are of three types:

(a) decimal integers: one or more decimal characters possibly terminated with the letter D.

2129, 600D, -217

(b) octal integers: one or more octal characters possibly terminated with the letter B and optionally a single-digit octal scaling factor.

217, 32B, 4B3 (which is the same as 4000_8)

(c) string: '1-4 characters (except ')'

All constants are absolute, i.e., their relocation value is 0.

The assembler normally expects integers to be decimal. This can be changed, however, by using a directive (OCT or DEC). In any case, integers may be terminated with B or D, overriding the normal interpretation of integers. String constants are not normally useful in the direct computation of memory addresses, but exist basically to be used in literals (cf. 5.0).

4.3 Classification of Symbols

The assembler recognizes the following types of symbols:

(a) local symbols: These symbols are defined by their use in the label field of instructions and in some directives. Their value is that of the location counter at their definition. They are thus symbolic addresses of memory cells. These symbols are relocatable (R = 1) if the assembly is relocatable; if the assembly is absolute, they are absolute. Once having been defined, a local symbol may not be redefined. Attempts to do so are considered errors, and diagnostics result.

(b) equated symbols: Equated symbols may be defined by equating them to an expression (using directives EQU, NARG, or NCHR). Their relocation value will be that of the expression. Unlike local symbols, equated symbols may be given new values at any point in the program.

(c) current location counter symbol (*): The character *, if used in the proper context, is understood to mean the current value of the location counter. It is relocatable or absolute depending on the nature of the assembly.

(d) external symbols: External symbols are those which are used but not defined in a given subprogram. They can be assigned no value, and it is not reasonable to regard them either as absolute or relocatable. External symbols may be used only as the sole object in an expression; other than its appearance as a sole object, the external symbol may not be used in an expression.

4.4 Terms

Terms are either constants or symbols, optionally preceded by a unary operator. The unary operator serves to modify both the value of the term and its relocation value. One unary operator -- special relocation, (R) -- may set the relocation value of a term to any value. This feature is explained in much more detail in 4.7.

4.5 Expressions

Expressions may consist of one or more terms connected by binary operators, or they may be just a single external symbol. Their evaluation proceeds from left to right using operators of decreasing precedence. For example, let A = 100, B = 200, and C = -1. Then

A+B*C/A = 98

Again, letting A = 54321_8, B = 44444_8, and C = 00077_8, then

A(OR)B(AND)C = 54365_8

4.6 Constraints of Relocatability of Expressions

The implementation of the assembler forces the following constraints on the use of expressions:

(a) No relocatable term (R = 1) may occur in conjunction with the operators * or /. In other words, no relocatable symbol may multiply, be multiplied by, divide, or be divided by anything.

(b) In the absence of the special relocation operator (R) the final relocation value of an expression may be only 0 or 1. It is possible that the relocation value may attain other values in the course of evaluation.

(c) If the special relocation operator (R) appears in an expression, then the relocation value of the expression may be either 0 or some other value K, where K is the special relocation radix. DDT is informed by the assembler that special relocation is being used in this case. DDT will then multiply the base address by K before adding it to the value of the expression (see next section).

4.7 Special Relocation

The special relocation feature has been provided to permit the programmer limited use of expressions which are not absolute or singly relocatable. To see why this is desirable, and how it works, consider the process of assembling and loading a relocatable program. Let the symbol A have value a. If one writes

```
        LDA   A
```

the assembler produces

```
        076   a
```

and marks the instruction's address as being relocatable. Later when told to load the program beginning at base address b, DDT will form

```
        076   a+b
```

Thus no matter where the program is loaded, the memory reference will be to the ath word from the base address.

Now suppose one writes

```
        LDA   2*A
```

The assembler, of course, can form

```
        076   2*a
```

and presumably what DDT should form is

```
        076   2*a+2*b = 076  2*(a+b)
```

To do this, it must be told that b is to be multiplied specifically by 2. Only one bit is reserved, however, for such information in the assembler's binary output; it is this fact which causes the restriction that expressions may have only the relocation values 0 and 1. And this restriction can be gotten around (inelegantly) by the use of (R). The following example gives one of the main reasons for which (R) was put into the assembler.

Programs may make use of the string-handling SYSPOPs of the 940. These instructions use string pointers, two-word objects containing starting and ending character addresses. Now characters are packed three per word. A character address therefore consists of the memory address containing the character multiplied by 3 plus 0, 1, or 2 depending on the position of the character in the word. If a character address is divided by 3, the quotient gives the word address and the remainder the character position in the word.

To form a character address at assembly time, one must be able to multiply a word address (a relocatable item) by a constant (in this case, 3). This is the reason for special relocation. The statement

```
        DATA     (R)A+1
```

will produce the value

```
        3*a+1
```

together with a notation to DDT that special relocation applies to that value.

DDT will then form the value

```
        (3*a+1)+3*b = 3*(a+b)+1
```

symbol, representing a relocatable word address, may thus be used to form character addresses in string pointers. There are other examples for the need for special relocation, but they will not be mentioned here. Let it suffice to say that special relocation is merely a device to make up partially for the rather severe relocation constraints the assembler imposes upon programmers.

It should be pointed out that the multiplicative constant associated with (R) in the example above was 3 because of the nature of string pointers. This constant is called the special relocation radix. It need not be 3 always. In fact, it may be changed to any value by the directive

RAD. Because of the relative importance of string pointers, however, the assembler is initialized with this value set to 3; it is hence unnecessary to use RAD to set it to 3 unless it has been changed for some reason.

5.0 Literals

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 produce automatically a location containing the correct constant in the program. Such a construct is called a literal.

Literals are of the form

=expression

When encountering a literal, 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 if 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 the above exception, the programmer need not concern himself with the locations of the literal values.

6.0 Directives

There is a large number of directives associated with this assembler. Although many of the directives are similar, each in general has its own syntax. A concise summary is given below:

Class	Directive	Use/Function
Data Generation:	COPY	Facilitates use of RCH command
	DATA	Generation of data
	TEXT	Generation of text
	ASC	Generation of text
Value Declaration:	EQU	Setting or changing symbol values
	EXT	Defining external symbols
	NARG	See
	NCHR	See
	OPD	Defining new op codes
	POPD	Defining pop codes
Assembler Control:	BES	Block ending symbol
	BSS	Block starting symbol
	ORG	Origin: absolute assembly
	END	End of program
	DEC	Interpret integers as decimal
	OCT	Interpret integers as octal
	RAD	Set special relocation radix
	FRGT	Forget name of symbol
	IDENT	Identify name of program
	DELSYM	Do not transmit symbols to loader
	RELORG	See 6.21
	RETREL	See 6.22
	FREEZE	Preserve symbols and macros
	NOEXT	Do not create external symbols
Output & Listing Control:	LIST	Set listing flags
	NOLIST	Reset listing flags
	PAGE	Skip to new page on listing
	REM	Type out remarks in pass 2
Macro Generation & Conditional Assembly:	MACRO	Head of macro body
	ENDM	End of macro body
	RPT	Begin repeat body
	CRPT	Begin conditional repeat body
	ENDR	End repeat body
	IF	Begin if body
	ELSF	Alternative if body
	ELSE	Alternative if body
	ENDF	End of if body

6.1 COPY Generalized Register Change Command

[[$]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 op codes 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	2	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
```

Of course, the symbols may be written in any order.

Clever programmers please note: This directive facilitates nicely some special RCH functions which might not otherwise be attempted (it is usually too much trouble). For example,

```
        COPY  AX,BX
```

has the effect of loading into X the logical OR (merging) of the A and B registers. Interested readers are referred to the SDS 940 manual for more details of the RCH instruction.

6.2 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
```

6.3 TEXT Generate Text

```
        [[$]label]  TEXT  'text'  [comment]
```

or,

```
        [[$]label]  TEXT  expression,text  [comment]
```

The TEXT directive is used to create a string of 6-bit trimmed ASCII characters, packed four to a word and assigned to increasing memory locations. The first word of the string may be labeled. The string to be packed may be delineated either by enclosing it in quotes (as in the first case above) or by preceding it with a word count (as in the second case). The second form of the directive must be used, of course, if the string contains one or more quotes. A potential hazard arising here should be pointed out. If a statement contains a single quote (or any odd number of them), it will not terminate with a semi-colon; a carriage return must be used.

```
        TEXT  4,THIS WON'T WORK; TEXT  4,DISASTER AHEAD
```

In the line above the semi-colon will be part of the text, and the second statement will be interpreted as being in the comment field,

```
        TEXT   4,THIS WILL '
        TEXT   1,A-OK
```

In the first form of the directive, characters in the last word are left-justified and remaining positions filled in by blanks (octal 00). In the second form, sufficient characters are packed to satisfy the word count.

6.4 ASC Generate Text with Three Characters per Word

This directive is identical in form and use to TEXT, except that 8-bit characters are packed three per word. The 940 string processing system normally deals with such text.

6.5 EQU Equals

```
    [$]symbol  EQU  expression    [comment]
```

The EQU directive causes the symbol in its label field to be defined and/or given the value of the expression. The expression must have a value when EQU is first encountered; i.e., symbols present in it must have been previously defined. It is permissible to redefine by EQU any symbol previously defined by EQU (or NARG or NCHR, cf. below). This ability is particularly useful in macros and conditional assembly.

6.6 EXT Define External Symbol

There are four ways which may be used to define external symbols.

(a) $label opcode or directive operand, etc.

The $ preceding the label causes the symbol in the label field to be defined externally at the same time it is defined locally.

(b) symbol EXT (comment not permitted)

The symbol given in the label field is defined externally. This symbol must have been defined previously in the program. The operand and comment fields must be absent.

Both of the above forms have the same effect; the name and value of a local symbol is given to the loader for external purposes.

Occasionally it is desirable to define an external symbol whose name is different from that of a local symbol; or an external symbol may be defined in terms of an expression involving local symbols. There are two ways of doing this.

(c) $symbol EQU expression [comment]

(d) symbol EXT expression [comment]

In (c) above the symbol is defined both locally and externally at the same time. (d) differs subtly in that the symbol in the label field is defined only externally; its name and value are completely unknown to the local program.

The feature (d) above is particularly useful in situations where two or more subprograms loaded together have name conflicts. For example, suppose programs A and B both make use of the symbol START, and A not only refers to its own START but B's as well. The latter references can be changed to BEGIN. Then into program B can be inserted the line

```
BEGIN  EXT  START
```

No other changes need be made either to A or B.

Occasionally, after having written a program, one would like to make a list of local symbols to be externally defined. A built-in macro ENTRY serves this function. That it is a built-in macro is irrelevant; the programmer may think of it as a related directive. Thus

```
          ENTRY   A,B,C,D,...
```

is precisely equivalent to

```
          A   EXT
          B   EXT
          C   EXT
          D   EXT
          .   .
          .   .
          .   .
```

6.7 NARG Equate Symbol to Number of Arguments in Macro Call

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

This directive may be used only in macro definitions. It is mentioned here only for completeness. It operates exactly as EQU except that in place of an expression in the operand field, the value of the symbol is set to the number of arguments used in calling the macro currently being expanded. Cf. 7.9 below.

6.8 NCHR Equate Symbol to the Number of Characters in Operand

```
          [$]symbol   NCHR   operand   [comment]
```

This directive is intended for use mostly in macro definitions, but it may be used elsewhere. It operates exactly as EQU except that in place of an expression in the operand field, the value of the symbol is set to the number of characters included in the operand field. A further explanation of the utility of this directive is deferred to section 7.

6.9 OPD Operation Code Definition

The OPD directive gives the programmer the facility to add to the existing table of operation codes kept in the assembler new codes or to change the equivalences of current ones. The form of OPD is:

```
opcode  OPD  expression,class[,ar[,type[,sb]]]  [comment]
```

where: 1) class must be 1 or 2 (cf. Section 3.1).

2) ar (address required) may be 0 or 1

3) type may be 0 or 1 (cf. Section 3.1).

4) sb (sign bit) may be 0 or 1

Quantities governed by the optional terms above (2,3 and 4) are set to zero if the terms are missing. As examples of how the directive is used, some standard machine instructions are defined as follows:

```
CLA     OPD     04600001B,2
LDA     OPD     76B5,1,1
RCY     OPD     662B4,1,1,1    (TYPE 1 = SHIFT)
```

A hypothetical SYSPOP LLA might be defined by

```
LLA     OPD     110B5,1,1,0,1
```

(class 1, address required, type 0, sign bit set).

In operation, the assembler simply adds new op codes defined by OPD to its opcode table. This table is always searched backward, so the new codes are seen first. At the beginning of the second pass the original table boundary is reset; thus if an opcode is redefined somewhere during assembly, it is treated identically in both passes.

6.10 POPD Programmed Operator Definition

In programs containing POPs it is desirable to provide the POPD directive. This directive works exactly like OPD and is used in the same way. Its essential difference from OPD is that it places automatically in the POP transfer vector (100_8 - 177_8) a branch instruction to the body of the POP routine.

In order to do this the assembler must know two things:

(1) the location for the branch instruction in the transfer vector and

(2) the location of the POP routine (i.e. the address of the branch instruction).

Item (1) is given by the POP code itself. Item (2) is provided by the convention that the POPD must immediately precede the body of the POP routine. The address of the branch instruction placed in the transfer vector is the current value of the location counter.

If the automatic insertion of a word in the POP transfer vector is not desired, then OPD should be used instead. An example of this case would occur in a subprogram containing a POP whose routine is found in another subprogram.

6.11 BES Block Ending Symbol

[[$]label] BES expression [comment]

The use of BES reserves a block of storage for which the first location after the block may be labeled (i.e. if the label is given). The block size is determined by the value of the expression; it must therefore be absolute, and it must have a value when BES is first encountered, (symbols present must have been previously defined). BES is most useful for labeling a block which is to be referred to by indexing using the BRS instruction (where the contents of X are usually negative). For example, to add together the contents of an array one might write:

```
      LDX  =-100     ARRAY HAS 100 ENTRIES
      CLA
LOOP  ADD  ARRAY,2   NEGATIVE INDEXING HERE
      BRX  *-1
      STA  RESULT
      HLT
ARRAY BES  100
```

6.12 BSS Block Starting Symbol

```
[[$]label] BSS expression [comment]
```

The use of BSS reserves a block of storage for which the first word may be labeled (if the label is given). The block size is determined by the value of the expression; it must therefore be absolute, and it must have a value when BSS is first encountered. The difference between BSS and BES is that in the case of BSS the first word of the block is labeled, whereas for BES the first word after the block is labeled by the associated symbol. BSS is most useful for labeling a block which is referred to by positive indexing (cf. 6.11 above).

6.13 ORG Program Origin

```
ORG expression [comments]
```

The use of ORG forces an absolute assembly. The location counter is initialized to the value of the expression. The expression must therefore be absolute, and it must have a value when ORG is first encountered. An ORG must precede the first instruction or data item in an absolute program, although it does not necessarily have to be the first statement. The output of the assembler will have a bootstrap loader at the front which is capable of loading the program after initiation by the 940 FILL switch.

6.14 END End of Assembly

```
END [expression]
```

The END directive terminates the assembly. For relocatable assemblies, no expression is used. For absolute assemblies the expression gives the starting location for the program. When assembling in absolute mode, the assembler produces a paper tape which can be read into the machine with the FILL switch, i. e., out of the time-sharing mode. If the expression is not included with the END directive, the bootstrap loader on this paper tape will halt after the tape has read in. Otherwise, control will automatically transfer to the location designated in the expression.

6.15 DEC Interpret Integers as Decimal

DEC [comments]

Integers terminated with B or D are always interpreted respectively as being octal or decimal. On the other hand, integers not terminated with these letters may be interpreted either as decimal or octal depending on the setting of a switch inside the assembler. The mode controlled by this switch is set to decimal by the above directive.

When the assembler is started this mode is initialized to decimal. Thus, the DEC directive is not really necessary unless the mode has been changed to octal and it is desired to return it to decimal.

6.16 OCT Interpret Integers as Octal

OCT [comments]

As noted in 6.15 above, this directive sets a mode within the assembler to interpret unterminated integers as octal. When the assembler is started this mode is initialized to decimal. Thus, the OCT directive must be used before unterminated octal integers can be written.

6.17 RAD Set Special Relocation Radix

RAD expression [comment]

As explained in 4.7 it is possible in a limited way to have multiple-relocated symbols. This action is performed when the special relocation operator (R) is used. The value of a symbol preceded by (R) is multiplied by a constant called the radix of the special relocation. The loader is informed of this situation so that it can multiply the base address by this same constant before performing the relocation. Because the special relocation was developed specifically to facilitate the assembly of string pointers (cf. 4.7), this constant is initialized to 3. If it is desired to change its value, however, the RAD directive must be used. The value of the expression in the operand field sets the new value of the radix. It must be absolute, and the expression must have a value when it is first encountered.

6.18 FRGT Forget Name of Symbol

FRGT $s_1,s_2,s_3,\ldots$ [comment]

where s_i are previously defined symbols

The use of FRGT prevents the symbol(s) named in its operand field from being listed or delivered to DDT. FRGT is especially useful in situations, for example, where symbols have been used in macro expansions or conditional assemblies. Frequently such symbols have meaning only at assembly time; they have no connection whatever with the program being assembled. When DDT is later used, however, memory locations sometimes are printed out in terms of these meaningless symbols. It is desirable to be able to keep these symbols from being delivered to DDT.

6.19 IDENT Program Identification

symbol IDENT [comment]

IDENT causes the symbol found in its label field to be 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 to the symbol in question.

IDENT statements are otherwise useful for editing purposes. They are always listed on pass 2, usually on the teletype.

6.20 DELSYM Delete Output of Symbol Table and Defined Op-codes

DELSYM [comment]

DELSYM inhibits the symbol table and opcodes defined in the course of assembly from being output for later use by DDT. Its main purpose is to shorten the object code output from the assembler. This might be especially desirable for an absolute assembly which produces a paper tape which is to be filled into the machine.

6.21 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 into core in some position, is destined to run from another position in memory. (It will first have to be moved there in a block.) This is particularly useful when preparing program overlays.

RELORG, like ORG, takes an absolute expression denoting some origin in memory. It has the following effects:

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

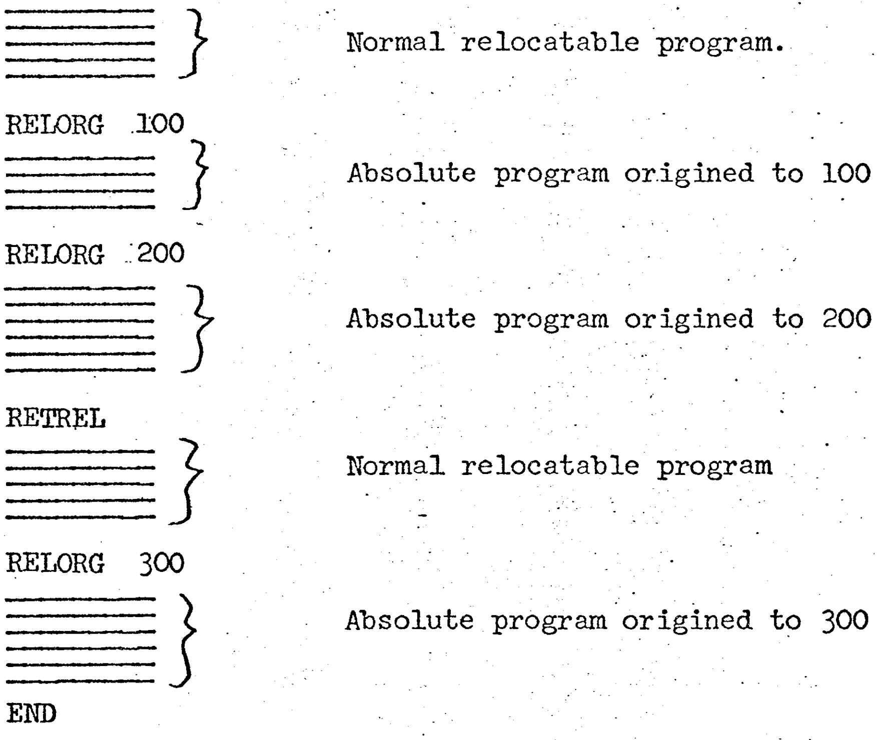
(b) The mode of assembly is switched to absolute without changing the object code format; it still looks like relocatable binary program to DDT. All symbols defined in terms of the location counter will be absolute. Rules for computing the relocation value of expressions are those for absolute assemblies.

It is possible to restore normal relocatable assembly (cf. 6.22, RETREL).

Some examples of the use of RELORG follow:

(1) A program begins with RELORG 300B and ends with END. 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. (It is, of course, necessary to move the program to location 00300_8 before executing it.)

(2) A program starts and continues normally as a relocatable program. Then there is a series of RELORGs and some RETRELs. The effect is as shown below:


Normal relocatable program.
RELORG 100
Absolute program origined to 100
RELORG 200
Absolute program origined to 200
RETREL
Normal relocatable program
RELORG 300
Absolute program origined to 300
END

6.22 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. The use of RETREL is shown in 6.21.

The effects of RETREL are

(a) to restore the location counter to what it would have been had the RELORG(s) never been used, and

(b) to return the assembly to relocatable mode.

6.23 FREEZE Preserve Symbols, Op-codes, and Macros

FREEZE [comment]

It is sometimes true when assembling various sub-programs that they share definitions of symbols, op-codes, 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. This greatly facilitates the editing of this information.

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 sub-program 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. one types

@ CONTINUE ARPAS.

Note that when the assembler has been pre-loaded with symbols, opcodes and macros, it cannot be released (i.e. one cannot use another sub-system like DDT, QED, etc.) without the loss of this information.

6.24 NOEXT Do Not Create External Symbols

Because of its subprogram capability, the assembler assumes automatically that symbols which are not defined in a given program are external and will be defined in another subprogram. It does not therefore list out the use of such symbols as errors.

If a program is in fact a free-standing program, i.e. if it is supposed to be complete, then clearly symbols which are not defined are errors and should be so noted in assembly. The NOEXT directive simply prevents external symbols from being established; thus undefined symbols are noted as errors. The directive must be used at the beginning of a program before instructions or data have been assembled. Its use affects the entire program. Its form is

```
NOEXT  [comment]
```

6.25 LIST Turn Specified Listing Controls On

6.26 NOLIST Turn Specified Listing Controls Off

Most assemblers provide a means of listing a program during assembly, i.e. printing out such items as the location counter, binary code being assembled, source program statement, etc. The association of these items on one page is frequently of great help to programmers. Two directives, LIST and NOLIST, control this process. Their form is as follows:

$$\left.\begin{array}{r}\text{LIST}\\\text{NOLIST}\end{array}\right\}\; s_1, s_2, s_3, \ldots \quad [\text{comment}]$$

where the s_i are from a set of special symbols having meaning only when used with these directives.

There are many listing options for this assembler. A list of special mnemonic symbols used in conjunction with these two directives is given below. The symbols have special meaning only when used with LIST and

NOLIST. They may be used at any other time for any particular purpose.

The special symbols are:

<u>Symbol</u>	<u>Meaning</u>
1	Listing during pass 1. Listing format will be controlled by other parameters.
2	Listing during pass 2. Listing format will be controlled by other parameters.
LCT	Listing of location counter value (see below)
BIN	Listing of binary object code or values (see below)
SRC	Listing of source language (see below)
COM	Listing of comments (see below)
MC	Listing of macro calls (see below)
ME	Listing of certain directives during macro expansions (EQU, NCHR, NARG, RPT, CRPT, ENDR, IF, ELSF, ELSE, ENDF, ENDM).
EXT	Listing of external symbols at end of assembly
NUL	Listing of null & duplicate symbols at end of assembly.

As an example of the meanings of various symbols above, consider the line of code   A21 STB OUTCHR   SAVE POINTER..

It might list as

$$\underbrace{\texttt{02157}}_{\texttt{LCT}}\quad \underbrace{\texttt{0 36 00217}}_{\texttt{BIN}}\quad \underbrace{\texttt{A21 STB OUTCHR}}_{\texttt{SRC}}\quad \underbrace{\texttt{SAVE POINTER}}_{\texttt{COM}}$$

It is not necessary to include each symbol possible, but rather only those parameters for which changes are desired. It is, in fact, not necessary to give <u>any</u> symbols.

```
          LIST   is equivalent to   LIST  2
```

When the assembler is started, it initializes itself in the following way:

```
      LIST    LCT,BIN,SRC,COM,MC,EXT,NUL
    NOLIST    1,2,ME,SYT
```

The actual format of the assembly listing is controlled by the current combination of parameter values. The parameters are independent items except for the parameters MC and ME. In this case it is more reasonable to think of their combination. Thus:

MC	ME	Effect
0	0	List outer level macro calls only
1	0	List all macro calls and code generated, but suppress listing of certain directives (see ME in table above).
0	1	List no macro calls, but rather all code generated except for certain directives.
1	1	List everything involved in macro expansions.

Regardless of the list control parameters which have been given to the assembler, it can be made to begin listing at any time in either pass simply by typing a single rubout (typing a second rubout in succession will abort the assembly). Listing having been started in this manner can be stopped by typing the letter S.

6.27 PAGE Begin New Page on Assembly Listing

```
      PAGE    [comment]
```

This directive causes a page eject on the assembly listing medium unless a page eject has just been given. It is used to improve the appearance of the assembly listing.

6.28 REM Type Out Remarks in Pass 2

```
      REM  remark to be typed
```

This directive, when encountered in pass 2, causes the contents of its operand and comments fields to be typed out either on the Teletype or whatever file has been designated as the output message device. This typeout occurs regardless of what listing modes 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 (this for debugging purposes).

7.0 Macros and Conditional Assembly

Assemblers with good macro and conditional assembly capability can have surprising power. This assembler features such capability. In this section the facilities for dealing with macros and conditional assembly will be discussed. Many examples will be given.

7.1 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. The opcode is the name of a binary machine command, and the macro name is the name of a sequence of assembly language statements.

EXAMPLE 7-1.

The 940 has an instruction for skipping if the contents of a specified location are negative, but none for testing the accumulator. SKA (skip if memory and accumulator do not compare ones) will serve when used with a cell whose contents mask all but the sign bit. The meaning of SKA used in this way is "skip if A positive." Thus a programmer will write

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

Programs, however, are more than likely to have a logical need for skipping if the accumulator is negative. In these situations the programmer 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). Let these operations be defined as macros.

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

Now -- more in keeping with the operations the programmer has in mind -- he may write

```
A22     SKAN
        BRU     POSCAS
         .
         .
         .
```

The advantages of being able to use SKAP or 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 has in mind. Programs written in this way tend to be easier to read. Note, incidentally, as shown above 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.

7.2 Macro Definition

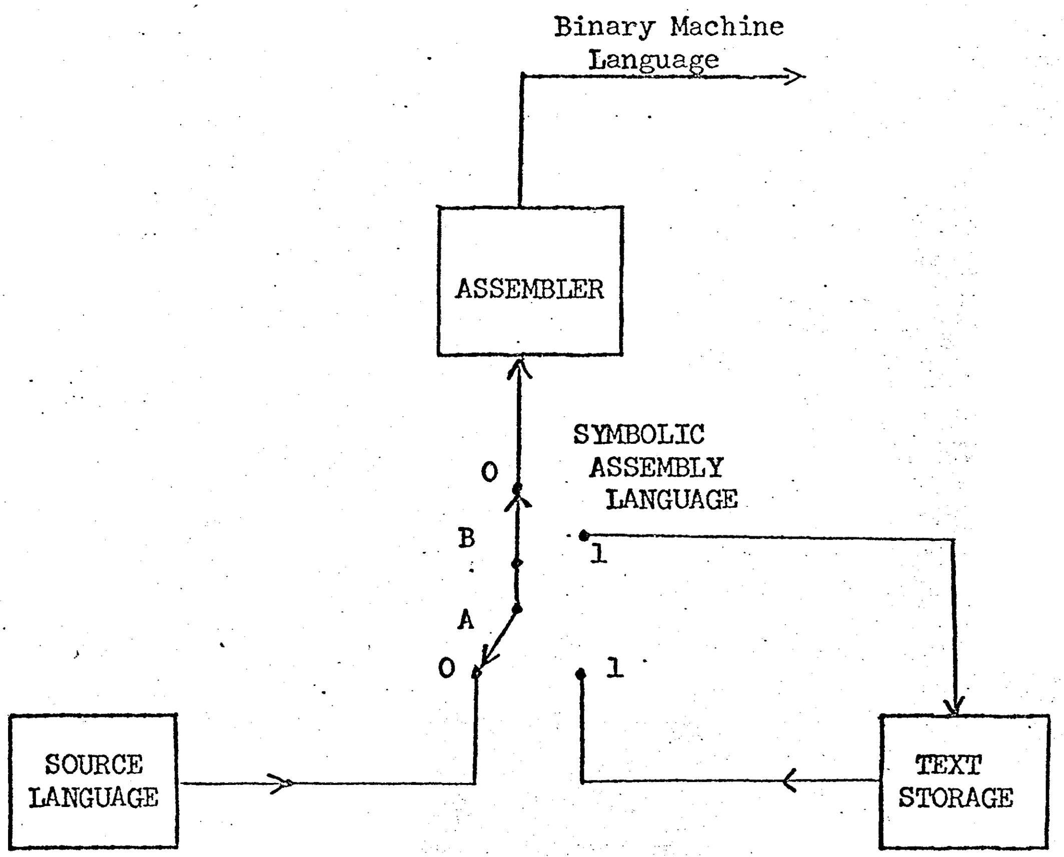
Before discussing more complicated use of macros, some additional vocabulary should be established. A macro is an arbitrary sequence of assembly-language statements together with a symbolic name. During assembly it is held in an area of memory called text storage. Macros may be created or defined. To do this one must give (1) a name and (2) the sequence of statements comprising the macro. The name and the beginning of the sequence of statements in a macro are designated by the use of the MACRO directive (see ex. 7-1 above).

```
            name    MACRO
                      .
                      .
                      .
                      .
                      .
                    ENDM
```

The end of the sequence of statements in a macro is signalled by the ENDM directive.

The reader should now refer to Figure 1. When the assembler encounters a macro definition (i.e., when it sees a MACRO directive), switch B is thrown to position 1. The programmer's source language is merely copied into text storage; note in particular that the assembler does not do any processing during the definition of a macro. Switch B is put back to position 0 when ENDM is encountered.

It is possible that within a macro definition other definitions may be imbedded. The macro defining machinery counts the occurrences of the MACRO directive and matches them against the occurrences of ENDM. Switch B is placed back in position 0 actually only when the ENDM matching the last MACRO is seen. Thus MACRO and ENDM constitute opening and closing brackets around a segment of source language. Structures like the following are possible:



A	B	Effect
0	0	normal assembly
0	1	macro definition
1	0	macro expansion
1	1	macro definition during macro expansion (to be explained in more detail later).

Figure 1: Information Flow During Macro Processing

```
name1  MACRO
         ⋮
name2  MACRO
         ⋮
name3  MACRO
         ⋮
       ENDM
         ⋮
name4  MACRO
         ⋮
       ENDM
         ⋮
       ENDM
         ⋮
name5  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 text storage in this case. What is important, however, is an understanding of _when_ the various macros are defined. In particular, when name1 is being defined, name2,3, etc. will _not_ be defined; they are merely copied unchanged into text storage. Name2 will not be defined until name1 is used*.

7.3 Macro Expansion

The use of a macro name in the opcode field of a statement is referred to as a _call_. The assembler, upon recognizing a macro call, moves switch A to position 1 (again see Figure 1). Input to the assembler from the original source language ceases temporarily and comes instead from text storage. During this period the macro is said to be undergoing _expansion_.

* It should be noted that macros -- like opcodes -- may be redefined.

It is clear that a macro must first 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 (when this makes sense). This is called recursion. Examples of the recursive use of macros are given later. When within a macro expansion a new macro expansion begins, information about the progress of the current expansion is put away. Successive macro calls cause similar information to be saved. At the end of each expansion the information about each previous expansion is restored in inverse fashion. When the final expansion terminates, switch A is placed back in position 0. Input then resumes from the source language program.

7.4 Macro Arguments

Now let us carry example 7-1 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.

EXAMPLE 7-2

Let us define the macro BRAP.

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

When called by the statement   BRAP   POSCAS

the macro will expand to give the statements

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

Note that BRAP was defined in terms of another macro SKAN (a matter of choice in this example). Also note that as defined, BRAP was intended to take only one argument. Other macros may use more than one argument.

EXAMPLE 7-3

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 called by the statement

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

the statements generated will be

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

Note that arguments furnished at call time are separated by commas. It is possible to include both commas and spaces in arguments by enclosing the arguments in parentheses; the macro processor strips off the outermost parentheses of any substring used in a call. For example in the call of the macro MUMBLE

```
MUMBLE A,(B,C),(D  E)
```

we have

```
D(1) =  A
D(2) =  B,C
D(3) =  D  E
```

7.5 The Use of Dummy Arguments in Macro Definitions

Before giving further examples of the use of macros, the various ways that dummy arguments may be used in macro definitions will be discussed. In general a dummy may be referred to by the symbolism

```
dummy(expression)
```

The only restriction on the expression above is that it must not contain other dummies or generated symbols (see 7.7). Furthermore, for obvious reasons it must have a known value when the macro is called*.

More than one dummy may be referred to by the notation

```
dummy(expression,expression)
```

In the case of the call

```
MUMBLE  A,B,C,D,E
```

then

```
D(3,5)= C,D,E
```

But it is possible to have confusion in this situation. If we have the call

```
MUMBLE A,B,C,(D,E),F
```

*It should be noted that a macro call may deliver more arguments than are referred to in its definition, but the converse is not true. A dummy argument not supplied with an argument at call time is considered an error.

then

DUM(3,5)= C,D,E,F

But which are DUM(3), DUM(4), and DUM(5)? To resolve this ambiguity, the assembler produces in place of DUM(3,5) the string

(C),(D,E),(F)

The notation

dummy()

produces all of the arguments supplied in a macro call. Each is surrounded by parentheses as in the example above.

The symbolism

dummy(0)

is legal and meaningful. It refers to the label field of the macro call. Normally a label used with a macro call is assigned the current value of the location counter (as with any instruction). Explicit use of dummy(0), i.e., literal zero in parentheses, causes the label field not to be handled in the normal way. It serves merely to transmit another argument. There are three possible cases.

(1) Macro contains no references to dummy(0). Label field is treated normally.

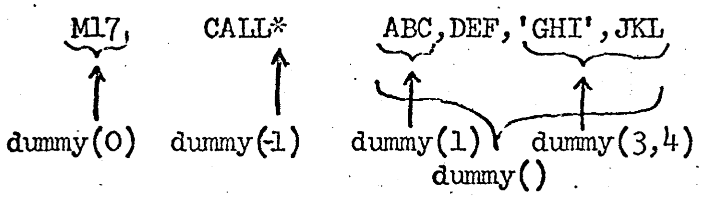
(2) Macro contains at least one reference to dummy(0). Label field merely transmits an argument which replaces dummy(0) in the expansion.

(3) Macro contains no references to dummy(0) explicitly but does contain dummy(expression) where, at call time, the value of the expression is zero. In this case the label field is handled as in case (1) and also used to transmit the argument referred to by dummy(expression) as in case (2).

The symbolism

dummy(-1)

is used to represent the terminal character of the opcode field, i. e., to determine whether the macro name terminated with a blank or a * (in case of indirect address). It allows macros to be called with or without "indirect addressing" specified. Thus in a typical call we have the following relationships:



Note that dummy(-1) is always one character long.

Sometimes in a macro definition it is desirable to refer only to a portion of an argument, perhaps to a character or a few characters. In the case of a single character this may be done by writing

dummy(expression$expression)

The first expression designates which argument; the second determines which character of that argument. If a substring of an argument is desired, one writes

dummy(expression$expression,expression)

The second and third expressions determine the first and last characters of the substring. For example, if we have the call

MUMBLE A,BCDE,'FGHIJ'

then

DUM(2 $3) = D

DUM(3 $4,7) = HIJ'

Beginning with the ith character the latter part of an argument can be obtained by specifying an overlarge terminal bound. Thus

```
DUM(2$4,1000) = HIJ'
```

7.6 Concatenation

It is frequently useful to compose statements out of macro arguments (or parts of them) and other information given in the macro definition. This is done by concatenating the various objects together, i.e. simply writing them next to each other. It is possible to confuse the assembler when doing this, however. For example, let the dummy name in a definition be C, and suppose we wish to concatenate the strings AB and C(3). If we write ABC(3), then do we mean AB concatenated with C(3), A concatenated with BC(3) (whatever that is), ABC(3), or what?

To avoid ambiguity we use the character "." (dot or period) as a concatenation delimiter. For the example just above we would write AB.C(3), and no ambiguity then exists. The assembler uses the dot to delineate objects it must deal with; in producing output the macro expansion machinery after having recognized the various objects simply skips over the dots. The dot character cannot therefore be used literally in a macro definition.

EXAMPLE 7-4

Let us define a macro STORE. Suppose we have established the convention that certain temporary storage cells begin with the letters A,B, or X, depending on from what 940 register information is to be stored there. The definition is

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

If called by the statements

```
STORE  B17
STORE* X44
```

the macro will expand as

```
                    STB    B17  or STX*   X44
```

The dot is not actually needed in every incidence of concatenation. Some programmers may readily determine for themselves when it is actually needed. As a matter of good practice, however, when in doubt, use it!

7.7 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. To do this requires two things; (1) the name and (2) the maximum number of generated symbols which will be encountered during an expansion. These two items may follow the dummy symbol name given in the MACRO directive. The actual format used is

```
     name   MACRO  dummyname,generatedname,expression
```

For example, we might have

```
     MUMBLE MACRO  D,G,4
              :
              :
            ENDM
```

In the definition of this macro there might be 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. Enough digits are produced to make the resultant symbol six characters long. Thus, the first time MUMBLE is called, for example, G(2) will be transformed into G00002, G(4) into G00004, 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 was 4 in the definition of MUMBLE above.) Thus the second call of MUMBLE will produce in place of G(2), G00006, the third call will produce G00010, etc. It should be clear that a generated symbol name should be kept as short as possible. It cannot be longer than 5 characters.

7.8 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 suitable 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 in value to the expression.

For example, let X = 5. Then

```
AB.($2*X-1)
```

will be transformed into

```
AB9
```

Further examples of the use of this facility appear below.

7.9 The NARG and NCHR Directives

Macros can be more useful if the number of arguments supplied at call time is not fixed. The precise meaning of a macro (and indeed, the results of its expansion) may depend on the number or the 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 basically like EQU, except that no expression is used with it. Its basic 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 required purpose. Examples of the use of NARG appear later.

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 characterstring    [comment]
```

The notion of "operand field" must be elaborated on here. The operand field normally terminates on the first blank after the beginning of the field. This rule is rescinded if a macro argument containing blanks appears in the operand field. For example, in the statement

```
XYZ    LDA  VECTOR,2    THIS IS A COMMENT
            ↑       ↑
```

the arrows delineate the operand field. Alternatively, if a statement like

```
TEXT X,D(1).ERROR
```

is placed in a macro definition and the macro is called by

```
MUMBLE  (NON-FATAL )
```

then the above statement will turn out to be

```
TEXT X,NON-FATAL ERROR
     ↑                ↑
```

Notice how the operand field terminates in this case.

In the same example notice that the message produced by the text directive is of unspecified length at definition time. Clearly, X must depend on the number of characters in D(1). Accordingly, MUMBLE might be defined as

EXAMPLE 7-5

```
MUMBLE MACRO  D
X      NCHR   D(1)
X      EQU    X+9  5 FOR 'ERROR',4 TO ROUND UP
       TEXT   X/4,D(1).ERROR
       ENDM
```

7.10 Conditional Assembly

The reader should see by now that the macro is a powerful tool. Its power, however, is considerably multiplied when combined with the features explained in this and the following sections. These features -- basically the if and repeat capabilities -- are called conditional assembly capabilities because they permit assembly-time calculations to determine the source language actually assembled. They are, however, not strictly a part of the macro facilities and may be used quite apart from macros.

7.11 The RPT Directive

The RPT (repeat) directive is, like the MACRO directive, an opening bracket for a segment of program. Its form is

(1) [label] RPT expression [comment]

or, using s for symbol, e for expression, and c for comment

(2) [label] RPT $(s=e_1,[e_2,]e_3)$ [c]

(3) [label] RPT $(s=e_1,[e_2,]e_3)(s=e_1[,e_2])(s=e_1[,e_2])\ldots$ [c]

Form (1) says to repeat the following sequence of statements down to the matching ENDR (end repeat) as many times as given by the value of the expression. Forms (2) and (3) are really the same form; they are shown separately to emphasize that only the first parenthesized group in the operand field must be present. Their meaning is as follows:

(1) Set the symbol s to the value of e_1.

(2) Issue the sequence of statements down to the matching ENDR.

(3) Increment s by the value of e_2 or by one (if e_2 is not present). If the new value of s has not passed the limit, go back to (2). When the limit is passed, quit.

In other words, for symbol=e_1 step e_2 until e_3 do ...

or for symbol=e_1 until e_3 do ...

The first parenthesized group (1) determines the number of times the repeat is executed and (2) controls the initial value and increment of a symbol. Subsequent groups (there may be up to ten of them) merely control the initial value and increments of other symbols carried along in the recent operation.

EXAMPLE 7-6

It is desired to create an area of storage which is cleared to zero. The BSS directive cannot be used for this purpose since its function (that of reserving storage) is basically to advance the assembler's location counter. The problem is readily solved by

```
ABC     RPT      100
        DATA     0
        ENDR
```

which is equivalent to

```
ABC     DATA     0                ↑
        DATA     0                |
        DATA     0                |
        DATA     0          100 statements
         .                        |
         .                        |
        DATA     0                ↓
```

Note that the label is applied effectively only to the first statement.

EXAMPLE 7-7

It is desired to fill an area of storage with data starting with 0 and increasing by 5 for each cell. We may write

```
X    EQU    0
     RPT    20
     DATA   X
X    EQU    X+5
     ENDR
```

Alternatively (and more simply) one can write

```
     RPT    (X=0,5,100)
     DATA   X
     ENDR
```

Note that in the latter form the terminal value (i.e., e_3) does not have to be positive or greater than the initial value of the symbol being incremented.

```
     RPT    (X=100,-5,20)
      :
and  RPT    (X=INIT,-5,-30)
      :
```

are both permissible.

Also note that a repeat directive followed by other statements and an associated ENDR (referred to as a repeat block) may be imbedded in other repeat blocks. This is similar to the imbedding of macro definitions in other macro definitions, and repeat structures similar to that shown in section 7.2 may be used.

EXAMPLE 7-8

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

```
SAVE    A,SUBRS
```

or perhaps

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

These calls are intended to generate the code

```
STA     SUBRSA
```

and

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

We first define a generalized macro MOVE which is called by the same arguments delivered to SAVE and RESTOR plus the strings 'ST' and 'LD' which determine whether one wishes to store or load.

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

Then, in terms of MOVE, SAVE and RESTOR are readily defined as

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

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

EXAMPLE 7-9

Many programs make use of flags, memory cells which are used as binary indicators. The SKN (skip if memory negative) makes it easy to test these flags. Let us adopt the convention that a flag is set if it contains the value -1 and reset if it contains zero. We want to develop the macros SET and RESET to manipulate flags. It is further desirable to deliver at call time the name of an active register which will be used for the action, together with a variable-length list of flag locations. Calls of these macros will look like

```
 SET     A,FLG1,FLG2,FLG3
```

or

```
RESET    X,FLG37,FLG12
```

As in example 7-8 we make use of an intermediate macro STORE which takes the same arguments.

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

Thus 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
```

7.12 CRPT, Conditional Repeat

Occasionally one wishes to perform an indefinite number of repeats, termination coming 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 block -- like RPT -- is closed off by a matching ENDR) except that instead of giving a number of repeats its associated expression is evaluated each time in a Boolean sense to determine whether the repeat should occur again. Its form is

[label]   CRPT   expression[,(s=e_1[,e_2]),(s=e_1[,e_2])...]
[comment]

One may write, for example,

```
          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 termination of a CRPT operation is governed by whether the value of the expression is one or greater. Zero or negative quantities are taken to mean don't repeat (Boolean 0 or false). Values of one or greater mean do repeat (Boolean 1 or true).

An example of the use of CRPT is shown in example 7-11.

7.13 IF Capability

It is frequently desirable to permit the assembler either to assemble or merely skip blocks of statements depending on the value of an expression at assembly time. This is primarily what is meant by the term conditional assembly. Conditional assembly can be done (inelegantly) with CRPT.

Let the condition be given by an expression. (Once again a Boolean value is ascribed to an expression in the manner

0 if $e \leq 0$

1 if $e > 0$.)

Then one may write

EXAMPLE 7-10

```
C       EQU      condition
        CRPT     C
         :
         :          arbitrary block of statements
C       EQU      0
        ENDR
```

Note that the line before ENDR is required to prevent the CRPT from going forever. By using the structure above, however, conditional assembly may be done; the arbitrary block of statements enclosed in the repeat body may be assembled on condition.

7.14 IF, Assemble if Expression True (i.e., > 0)

The same function shown in example 7-10 is performed much more conveniently by the IF directive. Its form is

```
[label]  IF    expression    [comment]
          :
          :
         ENDF
```

As with RPT and CRPT, the IF directive defines the beginning of a block of statements (called the if body) terminated by a matching ENDF. The if body may contain other if bodies.

When doing conditional assembly there are often alternative if bodies to be assembled in case a certain if body does not assemble. This situation is most easily dealt with by the use of the ELSF and ELSE directives. These provide an end to the if body and also begin another body which is to be assembled (again possibly on condition) in case the first body did not. For example, consider the following structure:

```
IF  $e_1$
  } $body_1$
ELSF $e_2$
  } $body_2$
ELSF $e_3$
  } $body_3$
ELSE
  } $body_4$
ENDF
```

If $e_1>0$, $body_1$ is assembled and $bodies_{2,3,4}$ are skipped (regardless of e_2 and e_3.

If $e_1\leq 0$ and $e_2>0$, $body_2$ is assembled and $bodies_{1,3,4}$ are skipped.

If e_1 and $e_2\leq 0$ and $e_3>0$, $body_3$ is assembled and $bodies_{1,2,4}$ are skipped.

Finally if e_1, e_2, and $e_3\leq 0$, $body_4$ is assembled.

An example of the use of IF (and other features) follows.

EXAMPLE 7-10

This example serves to illustrate several of the preceding features and also the power of macros used recursively. The macro MOVE is intended to take any number of pairs of arguments. The first argument of each pair is to be moved to the second. Each argument, however, may itself be a pair of arguments, which may themselves be pairs, etc.

We first define MOVE. Basically it extracts pairs of argument structures and transmits such a pair to another macro MOVE1.

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

We now define MOVE1. It calls itself recursively until it comes up with a single pair of arguments. Then it generates code.

```
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)     EQU       G(2)+1
U        EQU       G(1)
V        EQU       G(2)
         MOVE1     D(V),D(V+U/2)
         ENDR
         ENDF
         ENDM
```

Thus when called by the line

```
         MOVE      A,B
```

the code generated will be

```
         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
```

Finally 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
```

In this case the main call results in the call -

```
                    MOVE1   (A,B),(C,D),(E,F),(G,H)
```

MOVE1 calls itself by

```
                    MOVE1   A,B,E,F
```

and again:

```
                    MOVE1   A,E
```

where the first code is generated. Then we get

```
                    MOVE1   B,F
```

Recursion then pops up to the call

```
                    MOVE1   C,D,G,H
```

and so on.

EXAMPLE 7-11

The following example makes use of virtually every feature in the macro and conditional assembly machinery. It is presented as a demonstration of the power inherent in the use of macros but not as a practical tool (critics have justly termed it the world's slowest compiler). The macro COMPILE when called with an arithmetic expression for its argument produces assembly language which computes the value of the expression in a minimum number of steps (subject to the left-to-right scan technique used). COMPILE in turn calls a large number of other macros. Their functions are explained by comments in the text below:

The COMPILE macro itself merely initializes some variables and calls EXPAND where the more difficult work is done. J is the total number of characters in the expression. K is used to keep track of the recursion level on which the work is being done (EXPAND calls itself recursively when it sees an opening bracket [ ). AVAIL is the counter for available temporary storage. NPTR and PPTR are stack pointers for the operand and operator stacks respectively.

```
COMPILE MACRO D;J NCHR D(1);K EQU 0;AVAIL EQU 1;NPTR EQU -1;PPTR EQU -1
  EXPAND D(1);  ENDM
```

EXPAND initializes I, the current character pointer. It places the value zero on the operator stack (marking its beginning on the current level) and fetches the first operand. It then sets a switch (G(1)) and goes into a cycle of fetching operators (GETP) and operands (GETN). If the precedence of new operators is less than or equal to that of the previous operators, code is generated. Otherwise the information is stacked and the scan continued.

```
EXPAND MACRO D,G,1;I EQU 1;K EQU K+1; STACK 0,P; GETN D(1); SET G(1)
   CRPT G(1)
      IF I<J; GETP D(1$I)
      ELSE; OPTOR EQU 11; RESET G(1)
      ENDF
      ;PSTAK EQU PST.($PPTR)
      CRPT OPTOR/10<PSTAK/10+1; GEN D(1)
      ENDR
      IF OPTOR=11;PPTR EQU PPTR-1; RESET G(1);K EQU K-1;I EQU I.($K)+I-1
      ELSE; STACK OPTOR,P
          IF NPTR>0
              IF NST.($NPTR-1)<0
                  IF NST.($NPTR-1)=-1; STA TEMP.($AVAIL)
                  ELSE; RSH 1; STB TEMP.($AVAIL)
                  ENDF
                  ;NST.($NPTR-1) EQU AVAIL;AVAIL EQU AVAIL+1
              ENDF
          ENDF
          GETN D(1$I,J)
      ENDF
   ENDR
 ENDM
```

SET and RESET change the setting of flags. STACK is used to put values and pointers on "stacks." (These are not, of course, physical stacks in memory but rather conceptual ones existing in the assembler's symbol table.) STACK functions by creating an ordered progression of names and assigning values to the names by means of the EQU directive.

```
SET MACRO D;D(1) EQU 1; ENDM

RESET MACRO D;D(1) EQU 0; ENDM

STACK MACRO D;TS EQU D(2).PTR+1;D(2).PTR EQU TS;D(2).ST.($TS) EQU D(1)
 ENDM
```

GETN fetches the next operand. Its complexity is due to the fact that it must recognize symbols (in this example using the assembler's symbol rules) and numbers. When this recognition is complete it puts in the operand stack a pair of pointers to the head and tail of the operand (i.e., character numbers in the string and a flag bit which denotes whether the object is a symbol or a number. Note that if an opening bracket is seen, GETN calls EXPAND recursively.

```
GETN MACRO D;TO EQU I; RESET ERROR; GETC D(1$I-TO+1)
   IF CHAR='[';I.($K) EQU I; EXPAND D(1$2,J)
   ELSE
      IF LETTER; RESET NUMBER
      ELSE; SET NUMBER
      ENDF
      IF DIGIT; SET SWITCH
          CRPT SWITCH; GETC D(1$I-TO+1)
              IF DIGIT
              ELSF LETTER; RESET SWITCH
                  IF CHAR='B'; GETC D(1$I-TO+1)
                      IF LETTER; RESET NUMBER
                      ELSF DIGIT; RESET NUMBER
                      ENDF
                  ELSE; RESET NUMBER
                  ENDF
              ELSE; RESET SWITCH
              ENDF
          ENDR
      ELSF LETTER
      ELSE; SET ERROR
      ENDF
      IF NUMBER
      ELSE; SET SWITCH
          CRPT SWITCH; GETC D(1$I-TO+1)
              IF LETTER
              ELSF DIGIT
              ELSE; RESET SWITCH
              ENDF
          ENDR
      ENDF
      IF ERROR; ERROR; STACK 0,N
      ELSE; STACK TO*1B4+I-2+4B3*NUMBER,N
      ENDF
   ;I EQU I-1
   ENDF
 ENDM
```

GETC's main function is to determine whether a given character is a letter, digit, or other type of character. GETP fetches the next operator. It does some checking of the results and if valid sets OPTOR to a value carrying both operator and precedence information.

```
GETC MACRO D;CHAR EQU 'D(1)';I EQU I+1;A EQU CHAR>'Z';B EQU CHAR<'A'
   IF A(OR)B;A EQU CHAR>'9';B EQU CHAR<'0'
       IF A(OR)B; RESET LETTER; RESET DIGIT
       ELSE; SET DIGIT; RESET LETTER
       ENDF
   ELSE; SET LETTER; RESET DIGIT
   ENDF
 ENDM

GETP MACRO D; GETC D(1)
   IF LETTER(OR)DIGIT; ERROR
   ELSE;A EQU CHAR>11E6;B EQU CHAR<20E6
       IF A(AND)B;OPTOR EQU OPS.($CHAR/1E6)
       ELSF CHAR=']';OPTOR EQU 11
       ELSE;OPTOR EQU -1
       ENDF
       IF OPTOR=-1; ERROR;OPTOR EQU 40
       ENDF
   ENDF
 ENDM
```

GEN and GENA serve to reconstruct the operands from the string pointers and call generators which actually produce code.

```
GEN MACRO D;R EQU -1;PP2 EQU PST.($PPTR);PP3 EQU NST.($NPTR-1)
   ;PP4 EQU PP3/1E4;PP5 EQU PP3-PP4*1E4
      IF PP5>4E3;PP5 EQU PP5-4E3; SET LIT1; RESET LIT2
      ELSE; RESET LIT1; RESET LIT2
      ENDF
      IF PP3>1E4; GENA D(1),D(1$PP4,PP5)
   ELSF PP3>0; GENA D(1),TEMP.($PP3);AVAIL EQU PP3
      ELSF PP3=-1; GENA D(1),AREG
      ELSF PP3=-2; GENA D(1),BREG
      ENDF
   ;NPTR EQU NPTR-2; STACK R,N;PPTR EQU PPTR-1;PSTAK EQU PST.($PPTR)
 ENDM
```

```
GENA MACRO D;PP5 EQU NST.($NPTR);PP6 EQU PP5/1E4
   ;PP7 EQU PP5-PP6*1E4
   IF PP7>4B3;PP7 EQU PP7-4B3; SET LIT2
   ENDF
   IF PP5>1E4; GEN.($PP2) D(2),D(1$PP6,PP7)
   ELSF PP5>0; GEN.($PP2) D(2),TEMP.($PP5);AVAIL EQU PP5
   ELSF PP5=-1; GEN.($PP2) D(2),AREG
   ELSF PP5=-2; GEN.($PP2) D(2),BREG
   ENDF
 ENDM
```

GEN20, 21, 30, 31 and 40 are the code producing macros. They make reference to LIT1 and LIT2 (flags set by GEN and GENA) and call macros TEST, LA, LB, and ST. The purpose of the latter macros is to worry about the meaning of the contents of the A and B registers so as not to inject superfluous code.

```
GEN20 MACRO D; TEST D(1),D(2),X; LA D(X),LIT.($X)
   IF X=1
       IF LIT2; ADD =.D(2)
       ELSE; ADD D(2)
       ENDF
   ELSE
       IF LIT1; ADD =.D(1)
       ELSE; ADD D(1)
       ENDF
   ENDF
 ENDM

GEN21 MACRO D; TEST D(2),X
   IF X; LA D(2),LIT2
       IF LIT1; CNA; ADD =.D(1)
       ELSE; CNA; ADD D(1)
       ENDF
   ELSE; LA D(1),LIT1
       IF LIT2; SUB =.D(2)
       ELSE; SUB D(2)
       ENDF
   ENDF
 ENDM
```

```
GEN30 MACRO D; TEST D(1),D(2),X; LA D(X),LIT.($X)
   IF X=1
       IF LIT2; MUL =.D(2)
       ELSE; MUL D(2)
       ENDF
   ELSE
       IF LIT1; MUL =.D(1)
       ELSE; MUL D(1)
       ENDF
   ENDF
   ;R EQU -2
 ENDM

GEN31 MACRO D; TEST D(2),X
   IF X; ST D(2$1); LB D(1),LIT1; DIV TEMP.($AVAIL)
   ELSE; LB D(1),LIT1
       IF LIT2; DIV =.D(2)
       ELSE; DIV D(2)
       ENDF
   ENDF
 ENDM

GEN40 MACRO D; NOP D(1); NOP D(2)
 ENDM

LA MACRO D
   IF 'D(1)'='AREG'
   ELSF 'D(1)'='BREG'; LSH 23
   ELSE
       IF D(2); LDA =.D(1)
       ELSE; LDA D(1)
       ENDF
   ENDF
 ENDM

LB MACRO D
   IF 'D(1)'='BREG'
   ELSE
       IF 'D(1)'='AREG'
       ELSE
           IF D(2); LDA =.D(1)
           ELSE; LDA D(1)
           ENDF
       ENDF
   RSH 23
   ENDF
 ENDM

ST MACRO D
   IF 'D(1)'='BREG'; RSH 1
   ENDF
 ST.D(1$1) TEMP.($AVAIL)
 ENDM
```

```
TEST MACRO D;Y NARG;D(Y) EQU 0
   RPT (Z=1,Y-1)
       IF 'D(Z$1,4)'='AREG';D(Y) EQU Z
       ELSF 'D(Z$1,4)'='BREG';D(Y) EQU Z
       ENDF
   ENDR
   IF Y>2
       IF D(Y)=0;D(Y) EQU 1
       ENDF
   ENDF
 ENDM
```

The following lines establish precedence information for the arithmetic operators.

```
OPS10 EQU 30;OPS11 EQU 20;OPS12 EQU -1;OPS13 EQU 21;OPS14 EQU -1
OPS15 EQU 31
```

When called by the following lines, the macro generates code as shown:

```
Call:      COMPILE     X+200*Y

Result:    LDA =200
           MUL Y
           ADD X

Call:      COMPILE     AB-[C+D]/[E+F]

Result:    LDA      C
           ADD      D
           STA      TEMP1
           LDA      E
           ADD      F
           STA      TEMP2
           LDA      TEMP1
           RSH      23
           DIV      TEMP2
           CNA
           ADD      AB
```

```
Call:      COMPILE     A+200+34C21-[DEF/34B-HI*[J+20*K]/LM33B - N]/OPQ-22

Result:    LDA        =200
           MUL        34C21
           LSH        23
           ADD        A
           STA        TEMP1
           LDA        DEF
           RSH        23
           DIV        =34B
           STA        TEMP2
           LDA        =20
           MUL        K
           LSH        23
           ADD        J
           MUL        HI
           DIV        LM33B
           CNA
           ADD        TEMP2
           SUB        N
           RSH        23
           DIV        OPQ
           CNA
           ADD        TEMP1
           SUB        =22
```

7.15 Special Symbols in Conditional Assembly

Although in the introduction it is stated that symbols consist only of letters and digits, it is possible to include the colon in symbols. DDT, however, does not regard the colon as part of a symbol. The meaning of this is that DDT will type out such symbols but they cannot be typed in. In effect this makes them useless, and it is for this reason that the legality of colons in symbols has just now been mentioned.

Yet by judiciously choosing when to use the colon in a symbol the feature can become worthwhile. In particular it can be used in macros and other obscure places in the program to avoid possible conflicts with other names. This might be particularly useful to distinguish between symbols used in assembly-time calculations and those used at run-time.

8.0 Assembler Error Messages

Upon discovering an error in the syntax of a program being assembled, the assembler will list the statement in question and information about the character of the error. The listing of errors will occur regardless of whether regular listing is being done.

8.1 Error Messages

Error messages and their interpretations are given below. The first group deals with difficulties found in a single statement.

Error	Meaning
D	Duplicate symbol.
L	Error in label field; most likely not a valid symbol.
M	Missing field in statement.
O	Invalid or undefined opcode.
R	Relocation error in expression.
S	General syntax error.
U	Undefined symbol.

If when calling a macro the user fails to deliver an argument required during expansion, the assembler will replace the argument with the character ↑ and issue an undefined symbol message at that point.

The second group of error messages deal with more complicated difficulties

Error Message	Meaning
SYMBOL TABLE FULL. ERROR CHECK CONTINUES.	Too many symbols and/or opcodes have been defined. Assembly will continue, but no new symbols or opcodes will be recognized. Break the program into sub-programs or otherwise reduce the number of symbols present.

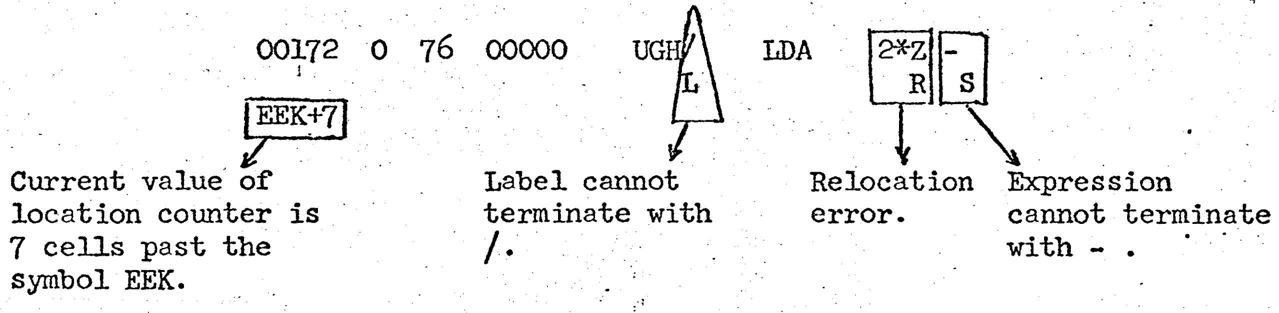
Error Message	Meaning
LITERAL TABLE FULL. FURTHER LITERALS IGNORED.	Similar to the case above. Reduce the number of literals present.
MUST ASSEMBLE ABSPGM ON PAPER TAPE	The bootstrap loader for self-filling, absolute assemblies is intended for paper tape only. Designating any other form on output file (except NOTHING and TELETYPE (another form of paper tape)) results in this message. It is possible to assemble an absolute program for loading by DDT. See 6.21 RELORG.
INPUT STACK OVERFLOW	There are too many nested macro calls, repeats, and ifs in combination. The stack provided for storing the previous source of input is full. This is a disaster. The program must be reorganized.
EOF -- END CARD ASSUMED	No END statement was found at the end of the program. The assembler (except for typing this message) takes the same action as if it found the END statement.
ILLEGAL COMMAND	The assembler does not recognize a command typed in by a user upon start-up. It makes him start again.
INPUT FILE NOT TEXT	The input file described to the assembler is not a type 3 file (i.e., text).
BAD CHAR	An unrecognizable character (or one otherwise out of place) is found in the text. The character is typed out in octal following the message, replaced by a blank in the text, and assembly continues.
EOF IN MACRO DEFINITION	The end of the program is reached, but the assembler is still defining a macro. Look for a missing ENDM.
INPUT STACK UNDERFLOW.	The opposite problem to the one above. Not terribly serious. Look for the presence of an extra ENDM, ENDR, or ENDF in the program.
INPUT BUFFER FULL.	An input statement must be less than 320 characters long. This message occurs when the rule is violated. It usually happens when macros run wild. Look carefully at the program near where the error occurred.

Error Message	Meaning
TOO MUCH MACRO RECURSION.	Too many nested macro calls have occurred, resulting in filling available pushdown storage. Reorganize program.
TOO MUCH RPT RECURSION.	Similar to above.
TOO MANY ARGS IN MACRO.	The macro is being called with more arguments than there is space for. Reduce the number of arguments in the call.
TOO MANY REPEAT ARGS.	In beginning a repeat block, too many requests for automatic incrementing of symbols have been made. Reorganize the block.
STRING STORE EXCEEDED.	No space remains to store new macro definitions or to do repeats. Caution: old macro definitions are not thrown away. Do not redefine macros indiscriminately. Reorganize program.
EOF IN TEXT.	The end of the input file has occurred in the middle of a statement.

8.2 Interpretation of the Error Listing

When an error is listed on any file other than TELETYPE, the single-letter error message (first group above) is listed in the line below at the point where the error was detected. Other information is given. This is all depicted in the examples below.

In the following line there are errors in the label and operand fields.



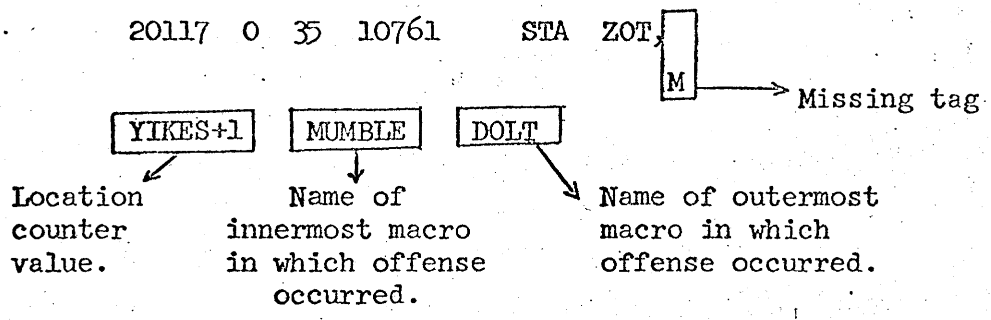
```
00172  0  76  00000    UGH/     LDA    2*Z  -
 EEK+7                   L                R  S
```

Current value of location counter is 7 cells past the symbol EEK.

Label cannot terminate with /.

Relocation error.

Expression cannot terminate with - .



```
20117  0  35  10761     STA  ZOT,  M
      YIKES+1  MUMBLE  DOLT
```

Missing tag
Location counter value.
Name of innermost macro in which offense occurred.
Name of outermost macro in which offense occurred.

Thus along with each error the location counter is printed out relative to the symbol most recently defined. In addition, if the error occurs during macro expansion the names of the innermost and outermost macros are printed to give a clue on where to look for the error. If only one level of macro expansion is involved, then only that name is listed.

In order to save time when error listings are made on the teletype, the single-letter error messages are typed out at the left margin.

9.0 Assembler Operating Instructions

When the assembler is entered it asks first for the name of the binary output file (characters typed by the user are underlined).

```
@ARPAS.
BINARY:
```

The user types the name of the binary file and confirms this name either with a comma or a period (a comma seems to the author to be preferable; both will work). The assembler then asks for the name of the source language file.

```
@ARPAS.
BINARY: 'BIN', INPUT:
```

The user types the input file name and confirms it either with a comma or period. Different things happen in this case. If a period is typed assembly begins immediately with the various assembler parameters preset to a set of standard initial values. If a comma is typed the assembler expects to see one or more of a number of commands which set its parameters to values other than standard initial values. Each such command is terminated either by a comma (in some cases a semi-colon) or a period. Assembly always begins when a period is typed; a comma signifies yet another parameter change.

9.1 Assembler Parameters

The assembler makes use of the parameters listed below. Parameters (a) and (b) must be supplied. If after this a period is supplied, default or standard values are given to the other parameters. These values are noted in parentheses.

(a) binary output file (specified by user)

(b) source input file (specified by user)

(c) list (text output) file (user teletype)

(d) error message file (user teletype)

(e) listing parameters

(1) list on pass 1 (no, except for disastrous errors)

(2) list on pass 2 (no, except for errors)

(3) list symbol table (no)

(4) list null symbols (yes)

(5) list external symbols (yes)

(6) list location counter value (yes)

(7) list binary (octal) equivalent (yes)

(8) list source language (yes)

(9) list comments (yes)

(10) list program and outer level macro calls (no)

(11) list program and all macro calls (yes)

(12) list program and no macro calls (no)

(13) list program and details of macro expansions (no)

Parameter (c) may be changed from its default value by the command "TEXT OUTPUT:". Thus:

```
@ARPAS.
BINARY:  BIN, INPUT: SIN, TEXT OUTPUT: L.
```

Normally, most error messages are output on the listing file. Some particularly disastrous errors are always typed out on the teletype. It is possible, however, to specify the teletype as a binary output file in which case it is used to punch a binary paper tape. It is necessary in such cases to designate some file --parameter(d)-- to receive those error messages which would normally appear on the teletype. This is done with the command "TTY MSG OUTPUT:".

```
@ARPAS.
BINARY: TELETYPE, INPUT: SIN, TTY MSG OUTPUT: E.
```

Binary paper tapes punched on the teletype may be read with the 940 paper tape reader. The assembler pauses for a few seconds after the final period is typed to allow the user to turn on the teletype paper tape punch and turn off the printing mechanism.

The various parameters noted under (e) are the same parameters set or reset by the LIST and NOLIST directives. They may be changed from their default values by the similar commands "LIST:" and "NOLIST:". Either command expects to see one or more symbolic arguments (the same arguments used with the corresponding directives). Because these arguments are separated by commas, it is necessary to terminate the list either by a period (starting assembly) or a semi-colon (if other parameters are yet to be set).

```
@ARPAS.
BINARY: NOTHING, INPUT: SIN, LIST: ME;
NOLIST: BIN,NUL,EXT;
TEXT OUTPUT: L.
```

The following table summarizes the above information.

Parameter	Command to Set or Change	Default value
binary output file	BINARY:<file name>	none: must be specified by user.
source input file	INPUT: <file name>	none: must be specified by user.
listing file	TEXT OUTPUT: <file name>	teletype
error message file	TTY MSG OUTPUT: <file name>	teletype
listing controls	LIST: }<list of following NOLIST: } parameters>	
	1	no
	2	no
	SYT	no
	NUL	yes
	EXT	yes
	LCT	yes
	BIN	yes
	SRC	yes
	COM	yes
	MC	yes
	ME	no

9.2 Termination of the Assembly

The assembly process can be terminated by the user at any time merely by typing two or more rubouts in succession. The first rubout causes the assembler to start listing (cf. 6.26).

10.0 Assembler Binary Output*

There are two basic formats for the assembler output. These are selected depending on whether relocatable or absolute assembly is being done.

10.1 Relocatable Binary Output

Information in this type of output is divided into variable length logical records. Each record begins with a control word which defines its type. The first nine bits (bits 0-8) of each control word distinguish it from the others; the remaining bits are used in various ways.

Control Word	Meaning and Use
(a) 000 XXXXX	Binary program follows. Update location counter by amount given in address field of control word.
(b) 1XX 00000	Programmed operator follows. Place branch instruction in location 1XX with address given by current location counter.
(c) 200 00000	End of program. Final record of binary format.
(d) 201 XXXXX	Origin of literal table. The origin of the literal table is given in the address field.
(e) 202 XXXXX	Change special relocation radix. The new value is given in the address field.
(f) 300 00000	OPD follows. Revert to triplet format (see below).
(g) 400 00000	External symbol definition(s) follows. Revert to triplet format.
(h) 500 00000	Identification record follows. Revert to triplet format.
(i) 600 00000	External symbol usage table follows. Revert to triplet format.
(j) 700 00000	Symbol table follows. Revert to triplet format.

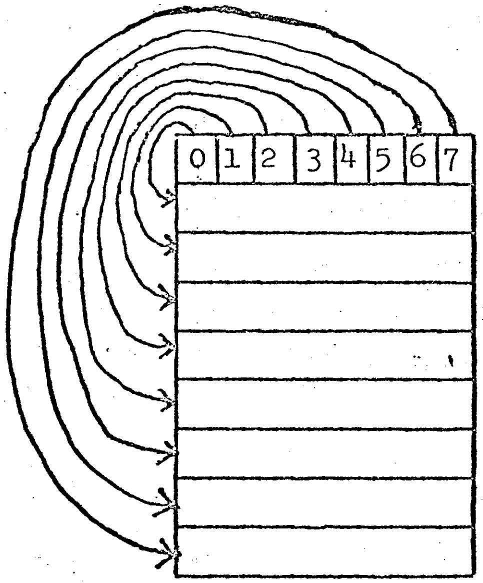
Control words b, c, d and e cause DDT to take various actions. No additional information is required for these controls, however; each is complete in itself. This is to be contrasted to controls f through j. Each of the latter prepares DDT to accept a variable-length list of symbols or opcodes.

*For a more complete description of binary input to DDT see Project Genie

These lists are in so-called "triplet format" because the various symbols and opcodes are handled as three-word objects. Each list is terminated with a word of all ones.

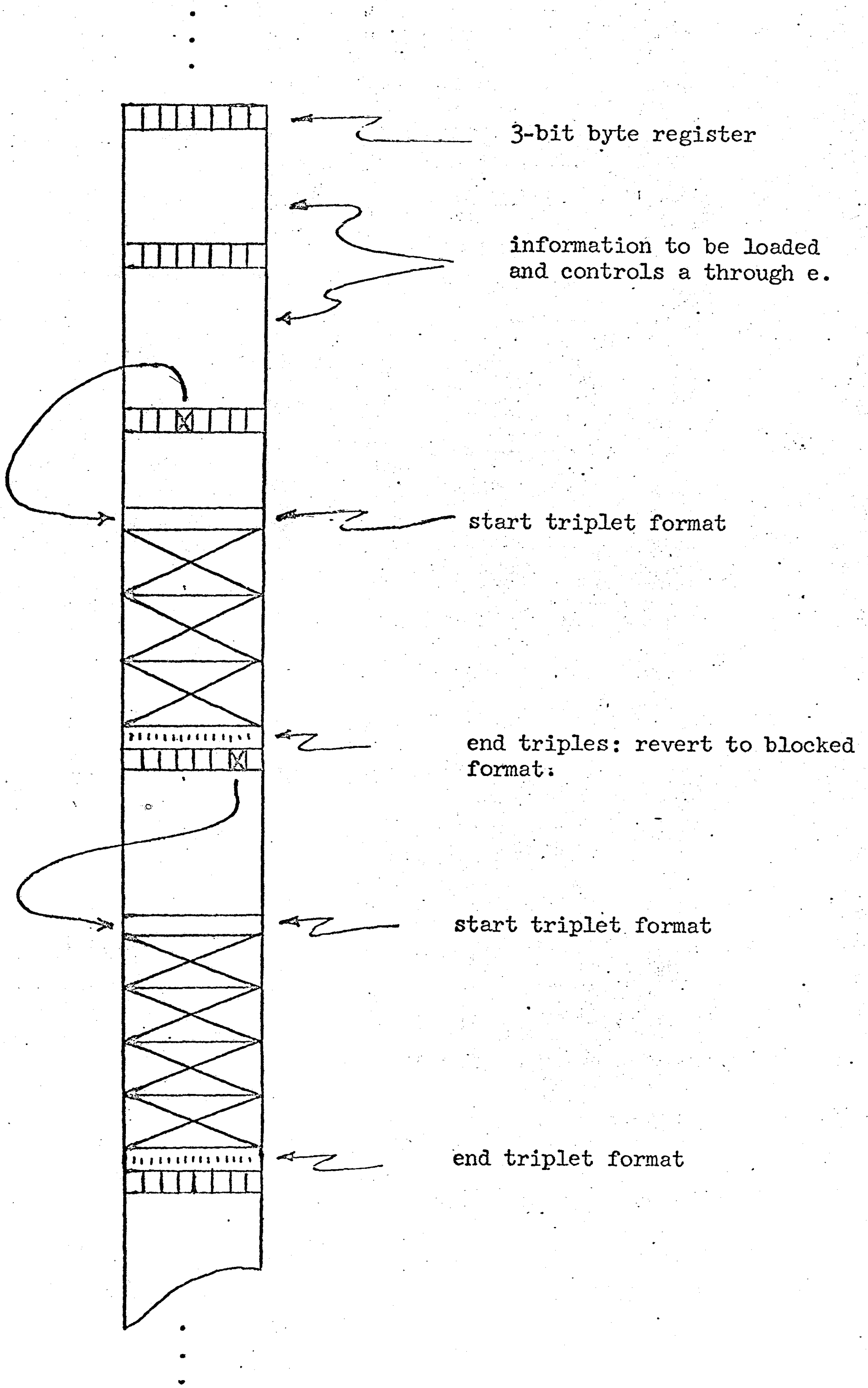
The contents of the address field of control word a are added to DDT's location counter. This control signals that binary program (i.e. information to be loaded) follows. The format of binary program consists of blocks of eight words. Words in each block are either loadable information or controls of type a through e. Controls f through j also appear in binary program; when one of them does, however, format immediately reverts to triplet mode. When the list of triples is terminated a new block of eight words is begun. The first word in this block is always a control of some type.

There are seven different ways in which DDT treats information being loaded. It is necessary, therefore, that a three-bit byte be associated with each such word. Each eight-word block of binary program format is preceded, then, by a word of eight three-bit bytes. The association of bytes to words is shown below.



Byte Value	Meaning
0	Absolute address: load as is
1	Evaluate address (mod 2^{14}) from external symbol usage table
2	Relocate address (mod 2^{14})
3	Special relocation applies
4	Do not load: interpret word as a control
5	Derive entire word from external symbol usage table
6	Relocate entire word (mod 2^{24})
7	Literal reference in address field

A portion of binary output might, for example, have the following appearance:


3-bit byte register
information to be loaded and controls a through e.
start triplet format
end triples: revert to blocked format:
start triplet format
end triplet format

Finally, the format of a triple depends on whether it represents a symbol or op code. For symbols the following holds:

	0 5	6 11	12 17	18 23
WD1	C_1	C_2	C_3	C_4
WD2	C_5	C_6	xxxxxx	zeros
WD3	Symbol Value			

C_1 through C_6 are the six significant characters of the symbol, left-justified with trailing blanks. Bits 12 through 17 of WD2 are flags having the following meanings:

Bit	Meaning
12	Relocatable Symbol
13	Duplicate Symbol
14	External Symbol
15	Null Symbol
16	Generated Symbol
17	Equated Symbol

Operation codes have the following format:

	0 5	6 11	12 17	18 23
WD1	C_1	C_2	C_3	C_4
WD2	C_5	C_6	Zeros	

The format of WD3 depends on the type of op code. The various possibilities are:

(a) Class 1 instructions

	0	1 8	9		19		23
WD3	0	op or pop code	X	zeros	X	zeros	X

Bit	Meaning
9	Set sign bit of instruction
19	Operand required
23	Type number (0 or 1)

(b) Class 2 instructions

	0	1 8	9	10 23
WD3	1	op or pop code	X	remainder of code

Bit	Meaning
9	Set sign bit of instruction

10.2 Absolute Assembly (Self-filling) Output

For absolute programs, the assembler first punches a bootstrap loader and then program in variable-length blocks. A new block is begun whenever it is necessary to jump the location counter ahead by more than one or when certain directives are encountered. The bootstrap loader is quite a primitive one. Basically it executes repeatedly the first instruction it sees in each block of code while incrementing the X register. For program blocks the first instruction in each block is an indexed store.

For other types of blocks (as, for example IDENT records, OPD's, or the entire symbol table) the first word of each block is a NOP. END with a blank operand produces a new block beginning with HLT. In case the operand is not blank, the last record begins with a transfer to the location indicated. When this instruction is read by the loader and executed, the loaded program is started. The loader uses location 1 for this purpose so that the program may be easily restarted by pressing the machines' Start button and throwing the Run-Step switch to Run.

APPENDIX B

TABLE OF TRIMMED ASCII CODE FOR THE SDS 930*

(NUMERIC ORDER)

0	SPACE	31	9	62	R
1	$\boxed{!}$	32	:	63	S
2	"	33	;	64	T
3	$\boxed{\#}$	34	<	65	U
4	$	35	=	66	V
5	$\boxed{\%}$	36	>	67	W
6	$\boxed{\&}$	37	?	70	X
7	'	40	$\boxed{@}$	71	Y
10	(	41	A	72	Z
11	)	42	B	73	[
12	*	43	C	74	$\boxed{\backslash}$
13	+	44	D	75	]
14	,	45	E	76	$\boxed{\uparrow}$
15	-	46	F	77	←
16	.	47	G	144	EOT
17	/	50	H	145	WRU
20	0	51	I	146	RU
21	1	52	J	147	BELL
22	2	53	K	152	LF
23	3	54	L	155	CR
24	4	55	M		
25	5	56	N		
26	6	57	O		
27	7	60	P		
30	8	61	Q		

*The Teletype characters enclosed in boxes cannot be handled by ARPAS and are converted to blanks when present.