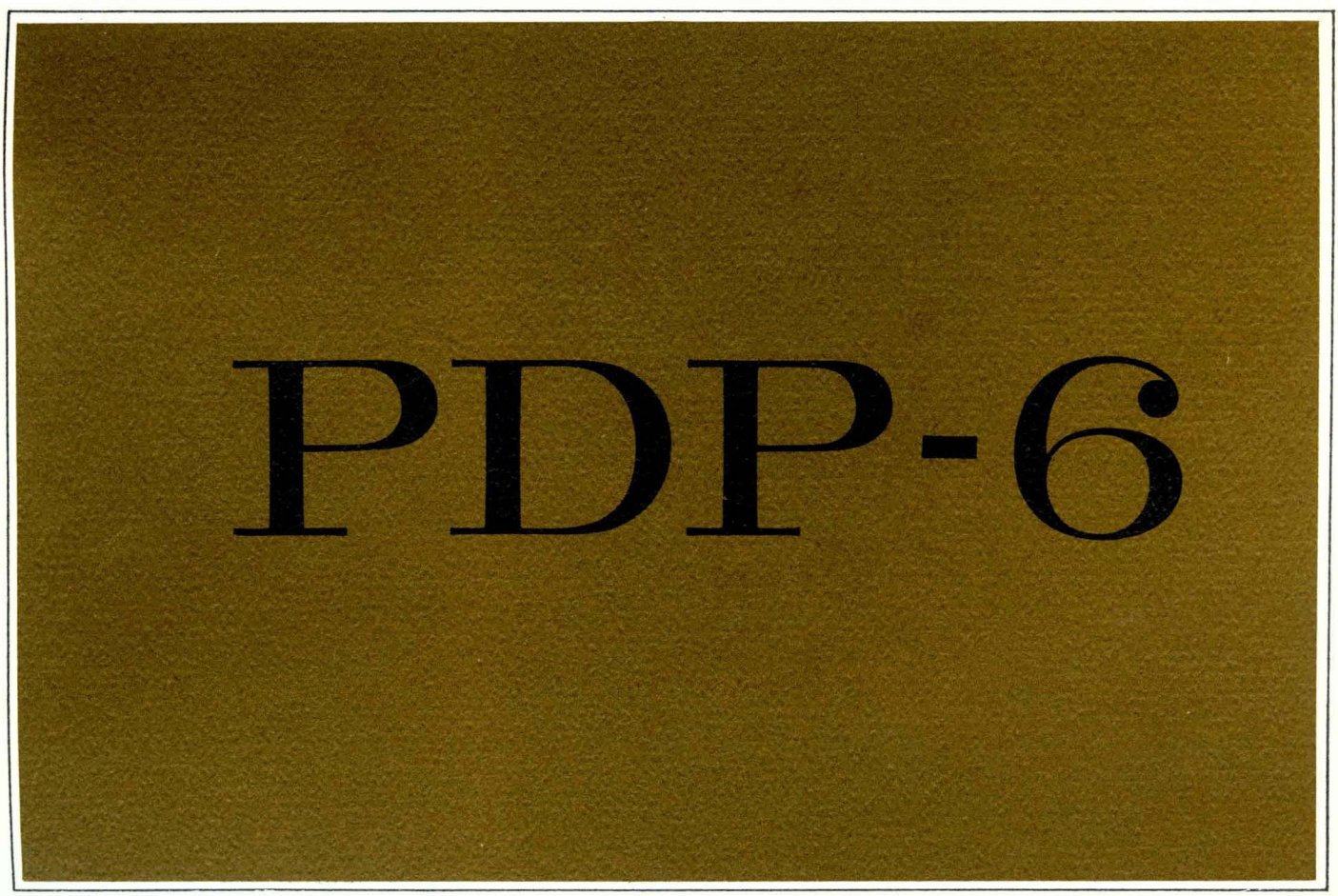
F-64 MAS
5/64

MACRO6
ASSEMBLY PROGRAM



DIGITAL EQUIPMENT CORPORATION • MAYNARD, MASSACHUSETTS

MACRO6

ASSEMBLY PROGRAM

CONTENTS

SECTION 1

INTRODUCTION

The use of an assembly program has become a standard practice in the programming of digital computers. This type of processor permits a programmer to code in a more convenient language than the 36-bit binary numbers which are of significance to a PDP-6; the processor translates the programmer's source language to machine code. The advantages of this are widely recognized: Easily recognized mnemonic codes are used instead of numeric codes; instructions or data may be referred to by a symbolic name without knowing or even caring about the actual machine address; decimal or alphabetic data may be expressed in a more convenient form than in a binary number; programs may be altered without extensive changes in the source language; and debugging is considerably simplified.

MACRO6 is an advanced type of assembly processor with a facility for using macro-instructions (where long sequences of code may be replaced by relatively short statements). It offers all the flexibility permitted by numeric coding with all of the above advantages, and in addition permits the assembly of relocatable programs.

The MACRO6 system consists of two parts: the source language in which programs are coded, and a computer program to translate this source language into PDP-6 code. This document describes the language and some of the details of the operation of the processor program.

The processor program is a two pass assembler in which the source language statements are processed twice, once to establish symbol definitions, and once to assemble PDP-6 code using these definitions.

The MACRO6 processor, like other parts of the PDP-6 Modular Software System is designed to reflect the modularity of the PDP-6 hardware.

The processor receives a string of characters as input. The output consists of a character string for listing and a string of binary words containing the assembled code. The processor interfaces with output and input routines which fetch and store these words and character strings. It is essentially immaterial whether the media containing the output and input strings are paper tape, punched cards, magnetic tape, Microtape, Teletype or a line printer. Core storage may even be used as an I/O medium on the larger configurations.

The processor produces relocatable binary code. This, too, is independent of the storage medium, in logical blocks containing information to be stored in memory by the linking loader. The assembled code format is completely compatible with all parts of the Modular Software System, including DDT-6, Checkpoint 6, the stack monitor and the time-sharing monitor.

SECTION 2

LANGUAGE FUNDAMENTALS

THE STATEMENT

The fundamental unit of the MACRO6 assembly language is the statement. A MACRO6 statement is a string of characters; it may be used to generate PDP-6 code, data, or to control the operation of the MACRO6 processor. A statement may be delimited by a semicolon or by a carriage return or the end of a card, depending on the input medium.

Each statement may be numbered. If the first character of a statement is a digit, all of the following characters up to the first blank are considered to be part of the statement number.

A statement is subdivided into fields which identify the data generated by the statement, specify the type of statement, describe the specific function of the statement, and comment the statement.

If a statement is ended with fewer fields than are normally required, the unspecified fields are considered null. If a statement has more fields than are required, the superfluous fields are taken to be comments. The information between the semicolon and the end of card or carriage return is also taken as a comment.

Label Field - This is a single symbol referring to the memory location where the next data word or instruction would normally be placed. This field is always delimited by a colon.

Code Field - This field consists of a single word. It describes the type of statement and is a mnemonic representing a PDP-6 instruction, a type of data configuration or a processor control code. It is delimited by either a space or a comma.

Variable Fields - The function of these fields is determined by the code field. They may describe data, machine addresses, accumulators, processor control, index registers, etc. They may be delimited by commas, parentheses or angle-brackets, depending on their function in the statement.

Fields consist of elements, codes, expressions and macros. Elements are single "words" and represent numeric values; codes are also single words and describe statement functions; expressions represent numeric values and are strings of elements separated by combinatory operators. Macros are single words and stand for character strings. The following are elements:

A

TAX

6

4.3 E-6

"TEXT"

These expressions are combinations of the above elements:

TAX+6

"TEXT" &6

A/4.3E-6+TAX

The following are codes:

HLLE

DATAO

XWD

Character Sets

Punched Paper Tape

The ASCII character set (Appendix IV) is used to construct statements. Two characters may not be used: the reverse slash (\) and the left arrow (←).

(They are ignored by the processor). All carriage returns must be followed by a line feed, and all line feeds must be preceded by either a carriage return or another line feed. Spaces are used to delimit the code field, and may be freely used in other places for formatting; tabs are logically equivalent to spaces, and are properly translated to spaces on output listings.

Punched Cards

A modified Hollerith code (Appendix IV) is used for constructing statements on punched cards. As with paper tape, the reverse slash (\) may not be used. Only the first 72 columns are considered by the processor; the remaining 8 may be used for identifying information. Spaces may be freely used for formatting; there is no particular usage of card columns for this purpose. To skip lines, blank cards are inserted; these generate no information.

THE LOCATION COUNTER

In general, statements generate 36 bit binary words, which are placed into consecutive memory locations. The location counter is a register used by the MACRO6 processor to keep track of the next available location in memory. It is updated after processing each statement. A statement which generates a single machine instruction would update the location counter by 1; a statement which generates 6 data words would update it by 6. The location counter may be explicitly set by the LOC or RELOC codes.

ELEMENTS

Elements represent binary integers less than 2^{36}. There are five types of elements: symbols, numbers, characters, points and literals.

Symbols - These are strings of letters and numbers, the first of which must be a letter. Although a symbol may be any length, only the first 6 characters are considered and any additional characters are ignored; symbols which are identical in their first six characters are considered identical. A decimal point may appear in the character string of a symbol but may not be the first character.

Example:

```
X
A65
NUMERIC (EQUIVALENT TO NUMERI)
X.3B
HIGH.
N12345
```

Numbers -

A number is a string of digits. If the string contains a decimal point, it is evaluated as a floating decimal number and the digits are taken radix 1Ø. If the string does not contain a decimal point, the digits are assigned values according to the prevailing radix. (This prevailing radix is normally regulated by the RADIX code). If 8 were the prevailing radix the number 17 would have the value $17_8=15_{10}$. If 1Ø were prevailing, 17 would have the value $17_{10}=21_8$. The number 17.Ø would always have the value 2Ø542ØØØØØØØ since the decimal point denotes a floating decimal number. A number must always begin with a digit or a decimal point.

Occasionally it may be desirable to change the value of the radix for only one numeric element. This is done by the qualifier ↑ followed by a letter. Numbers are qualified in this manner to be Decimal, Octal, or Binary

Thus:

↑D17 = 17_{10}
↑Ø17 = 15_{10}
↑B1Ø1Ø = 10_{10} = 12_8

irrespective of the prevailing radix. These qualifiers have no further effect on the prevailing radix. Floating decimal numbers never consider qualifiers; any qualifier is ignored. The exponent parts of floating

decimal numbers may be further argumented by following the number by E±n, whereupon the number is then considered to be multiplied by 1Ø±n.

Example:

```
1.
10.0E-1
0.0001E4
.001E+3   (ALL = 201400000000)
1E10
```

A number may also be logically shifted left by following it by Br. The number is then shifted left so that the right hand (low order) bit is in position r (decimal) of the 36 bit computer word.

Thus:

```
03B35 = 000000000003
03B31 = 00000000060
03B17 = 000003000000
```

Point - The decimal point alone has a special meaning. It represents the current value of the location counter.

Example:

```
A:    JUMP .+6
,EQUIVALENT TO
A:    JUMP A+6
```

Character - If the first non-blank character of an element is a quote ("), the characters following it are assembled as their 6 bit ASCII representations: This element is terminated by a quote. If more than 6 characters are included within the quotes, only the right hand 6 are considered.

Example:

"AXE" is equivalent to 417Ø45
(This representation is useful with
immediate mode operations).

Literal - This type of element consists of any statement which will generate one word of machine code or data, surrounded by a pair of brackets. The appearance of a literal causes a cell containing the information generated by the enclosed statement to be reserved, usually at the end of the program. The element represents the address of this reserved cell. Literals may be nested to any reasonable depth.

Examples:

```
ADD 2, [DEC 65], DECIMAL LITERAL
FAD 1, [8.14], FLOATING POINT
MOVE 3, [ASCII .BYTES.]
XCT[XCT[XCT[ADD 2,X]](4)],NESTED
```

The last example generates the following constants:

```
LIT1:  XCT LIT2(4)
LIT2:  XCT LIT3
LIT3:  ADD 2,X
```

EXPRESSIONS

Expressions are strings of elements separated by arithmetic or Boolean operators. Expressions represent numeric values less than 2^{35} in magnitude. The value of an expression is calculated by first substituting the numeric values for each of the elements and then performing the operations. The allowable operations are:

Operator		Meaning
*	-	multiply
/	-	divide
+	-	add
-	-	subtract
&	-	and
!	-	inclusive or

When combining elements, the Boolean operations (and, ior) are performed first, from left to right. Then the multiplications are performed from left to right, and finally the additions and subtractions are performed. Division always truncates the fraction part. All arithmetic is performed modulo $2_{35} = 34,\ 359,\ 738,\ 368$.

For example, suppose the element

A represents the value 2_{10}

B represents the value 8_{10}

C represents the value 3_{10}

D represents the value 5_{10}, the expression:

A/B + A*C represents 6_{10}

B/A - 2*A-1 represents $-1_{10} = 777777777777_8$

A&B represents Ø

B+Ø1Ø$_{10}$+C represents 21_{10}

1 + A&C represents 3_{10}

EVALUATION OF SYMBOLS

The value represented by a symbol is assigned by one of three mechanisms:

(1) Label - If a statement begins with a symbol followed by a colon, the symbol is called a label. It is assigned the current value of the location counter.

(2) Direct Assignment - If a symbol appears on the left hand side of the = in a direct assignment statement, it is assigned a value equal to that value represented by the expression on the right hand side. A direct assignment statement has the form:

```
SYM = EXP,      COMMENT
```

where SYM is a symbol and EXP is an expression:

Example:

```
A = B+2
TSIZE = TEND-TSTART
K = 4
```

(3) Variable - If a symbol is followed by #, a storage cell is automatically reserved, (usually at the end of the program), and the symbol represents the location of this storage cell.

This is useful for defining a symbolic temporary storage location, for example: TEMP#, which reserves a cell whose address is represented by TEMP.

If a value is assigned directly, it may be altered by another direct assignment statement. If it is defined as a label or variable, it may not be altered.

SECTION 3

STATEMENTS

The meaning of a statement is determined by the code field. The code is a mnemonic word representing a machine operation, a data configuration, or a processor control code. Every statement must have a code. The code field is delimited by either a space, a comma, or a statement delimiter.

Although codes are similar in appearance to symbols, they should not be confused. For example, ADD may be either a symbol or a code; in context, however, it is unambiguous.

COMMENT STATEMENTS

A statement with a blank code is considered to be a comment. (Obviously this code must be delimited by a comma).

Example:

```
, THIS IS A COMMENT
```

INSTRUCTIONS

When the code field contains the mnemonic of a machine instruction concatenated with a mode mnemonic, a statement represents a machine instruction. There are two types of instructions - primary and I/O. Two statements are:

```
READ: DATAI PTR, @NUM(4)
SUM: ADD 2, TABLE (X3)
```

The first field of an instruction statement is a label. The single symbol in this field is given a value equal to the memory address occupied by the instruction. The next field is delimited by a space and is the code. If the next field is delimited by a comma, it is the accumulator in a primary instruction or the device number in an I/O instruction; otherwise it is an expression for the address referenced by the

instruction. If there is an accumulator field, then the following field is the address. The last field, enclosed by parentheses, is an expression representing the index register. The @ appearing in the address field indicates an indirect address.

The accumulator or device field may be left out, and the code delimited by the comma or a space. In this case the accumulator or device number is considered to be zero. If indexing is not used the index field may also be left out, and the address field delimited by a comma, carriage return or semicolon. A blank address field is considered to be zero. The following instruction statements are proper.

```
JRST .+3;
ADD 2,X
JUMP (4)
CONO 203,  ENABLE PC ON CH 3
```

Assembly

Instructions are assembled in the following manner: Each instruction code represents a 36 bit number. If it is a primary instruction code, the low order 4 bits of the value of the accumulator expression are ior'd into positions 9-12. The low order 9 bits of the value of the device expression of an I/O instruction are ior'd into positions 3-11. The low order 18 bits of the value of the address expression are ior'd into the right half of the instruction. If the indirect address symbol @ appears in the address field, a bit is placed into position 13. Finally, if the index register field exists, the lower 4 bits are ior'd into positions 14-17.

Numeric Codes

Numeric codes are considered to indicate primary instructions. The remainder of the statement is assembled as a normal instruction. If the numeric code is preceded by a minus sign, the twos complement of the number is taken; the minus sign is ignored for other codes. Character elements are considered to be numeric.

Example:

```
0A;    THE VALUE OF A IS IN THE RT HALF
270B8 2,X;    EQUIVALENT TO ADD 2,X
↑D65;    000000000101 IS GENERATED
-1;    777777777777 IS GENERATED
```

EXTENDED INSTRUCTIONS

For programming convenience, some extended codes have been devised. These represent a machine instruction combined with an accumulator or device number.

JEN ≡ JRST, - Jump and enable the PI system

HALT ≡ JRST 4, - Halt, then Jump

JRSTF ≡ JRST 2, - Jump and reset flags

JOV ≡ JFCL 8, - Jump on over flow and clear

JCRY0 ≡ JFCL 4, - Jump on CRY0 and clear

JCRY1 ≡ JFCL 2, - Jump on CRY 1 and clear

JCRY ≡ JFCL 6, - Jump on CRY1 or CRY0 and clear

JPC ≡ JFCL 1, - Jump on PC change and clear

RSW ≡ DATA I 0, - Read the switches

DATA GENERATING CODES

Several codes are used to indicate various types of data configurations. These codes describe the type of data to be generated. A label on a data generating statement refers to the first word assembled.

DEC - Decimal data - set the radix to 1Ø for this statement only and generate a word for each expression following the code. Expressions are separated by commas.

Example:

```
DEC 1Ø, 4.5, 3.1416, 6.ØØ3E-26, 3;
5 WORDS GENERATED
```

OCT - Octal data - similar to the DEC code, but the radix is temporarily set to 8

Example:

```
OCT -3, 2, 777, 4.1;THE 4TH ITEM IS FLOATING PT.
```

EXP - Expressions - The radix is unchanged. Each expression following the code generates one 36 bit data word.

Example:

```
EXP X, 4, ↑Ø65, HALF, B+362-A;
```

XWD - Transfer word - Two expressions follow this code which generates one data word. The low order 18 bits of the value of the first are placed in the left half word, and the low order 18 bits of the value of the second expression are placed into the right half.

Example:

```
ATOB:  XWD A,B;    TRANSFER FROM AREA A TO B
```

Z- Zero word - One word containing zeros is generated.

Example:

```
TEMP:   Z,    TEMPORARY STORAGE
```

IOWD - I/O transfer word used in the BLKI and BLKO instructions. Two expressions separated by commas follow this code which generates one data word. The left half of the assembled word contains the twos complement of the value of the first expression, and the right half contains the value of the second expression minus one.

Example:

```
INAREA:  IOWD 6, ↑D265;
ASSEMBLES AS 777772000377
```

POINT - Byte Pointer Word. The first expression indicates the byte size, the second indicates the address, and the third indicates the position of the right hand bit of the byte position. The indirect character @, and an index expression in parentheses may appear in conjunction with the address part. The local radix for the position and size expressions is always 1Ø, regardless of the prevailing radix.

Example:

```
STRING:  POINT 6,@N(4),5,
,POINTS TO THE LH CHAR
```

If the position expression is left blank, the position part will assemble as 44_8. On incrementing, the pointer will point to the left hand byte.

SIXBIT

Alphabetic Information. This code is used to generate characters in 6 bit ASCII code, pack them into 6 character words and place the words in sequential registers. The first non blank character following the code is the delimiter. Information is assembled from the second character until the first character is repeated. Only the printing characters of the ASCII code are assembled, except that line feeds are assembled as 74 (\).

Example:

```
NUMBR2    SIXBIT "2"
ALPHA:  SIXBIT /ALPHABETIC INFORMATION/
,EQUIVALENT TO
ALPHA:  OCT 411460504142, 456451430051;
        OCT 564657625541, 645157560000;
```

ASCII -

Alphabetic information. This code is similar to SIXBIT, but it packs words with the low seven bits of the full ASCII representation. All of the ASCII characters may be assembled under this code, and the normal injunction against the reverse slash (\ ) and back arrow (←) is relaxed.

Example:

```
ASCII .
.;A CARRIAGE RETURN AND LINE FEED
```

BLOCK -

Block of storage reserved. - The expression following the code indicates the number of cells to be reserved. The location counter is incremented by the value of the expression.

Example:

```
MATRIX:  BLOCK N*M
```

BYTE -

Byte strings. The first expression following this code is enclosed in parentheses and is the byte size. Subsequent expressions, separated by commas, are evaluated, truncated to the byte size, packed and assembled into sequential memory locations. If a byte cannot fit into a word, it is assembled as the first byte of the next word. The byte size may be altered in the middle of a word or a string by inserting a byte size expression in parentheses. The local radix for the size portion is always considered to be 10, no matter what value the prevailing radix may have.

Example:

```
RADIX 10
AX:  BYTE (6) 10, 4, 9, 1, 1, 3, 6
Q:   BYTE (15) 12, 3, 9,
STR: BYTE (6) 10, 4, 9 (12) "AB"
,EQUIVALENT TO
AX:  OCT 120411010103, 060000000000;
Q:   OCT 000400000300, 000110000000;
STR: OCT 120411004142;
```

PROCESSOR CONTROL CODES

These statements do not generate data or instructions, but control the operation of the processor.

REPEAT - This code causes a character string to be repeatedly processed. The code is followed by an expression whose value indicates the number of repetitions desired. This is followed by the string to be re-processed enclosed by angle-brackets.

Example:

```
ADDX:  REPEAT 3<ADD 6,X(4)
                ADDI 4,1>
,EQUIVALENT TO
ADDX:  ADD 6,X(4)
       ADDI 4,1
       ADD 6,X(4)
       ADDI 4,1
       ADD 6,X(4)
       ADDI 4,1
SQ:  REPEAT N <
     EXP .*.+SQ*SQ+1-2*.*SQ+2*. -2*SQ>
A TABLE OF SQUARES
```

The label of a repeat is placed on the first statement generated.

REPEATs may be nested to any reasonable degree:

```
REPEAT N+1 <MOVE 6, A(K)
REPEAT N    <ADD 6, (3)
             ADDI 3,L>
             MOVEM 6,A(K) >
```

IFn - Conditional assembly - An IFn code is followed by an expression, and a string of coding enclosed in angle-brackets. If the expression fulfills the condition indicated by n, the string is processed; if not it is ignored. The IFn codes are:

IFE Assemble if expression is zero

IFG Assemble if expression is positive

IFGE Assemble if expression is positive or zero

IFL Assemble if expression is negative

IFLE Assemble if expression is negative or zero

IFN Assemble if expression is non zero

IF1 Assemble if PASS 1 (no expression)

IF2 Assemble if PASS 2 (no expression)

Example:

```
IFZ   X-Y   <ADD Z, X;>
,ASSEMBLED ONLY IF X=Y ,
```

IFIDN - Conditional assembly on character strings. This is followed by three sets of angle-brackets. If the character strings enclosed by the first two sets of angle-brackets are identical, the coding within the third set is processed.

Example:

```
IFIDN <+> <+> <FAD 3,X>
,FAD 3,X; WILL BE PROCESSED
```

IFDIF - Conditional assembly on character strings. This is the converse of IFIDN and is similar in format. The coding within the third set of angle-brackets is processed if the two character strings differ.

RADIX - The prevailing value of the radix is controlled by this code. It is followed by a decimal number between 2 and 10 which then becomes the pre-

vailing radix. The radix may be changed at any point on the assembly; it is initially considered to be 8.

Example:

```
      RADIX 10,
   ,SET PREVAILING RADIX DECIMAL
```

LOC - This code changes the location counter to a value equivalent to the following expression. The block of coding following a LOC is assembled into the absolute locations, and any labels defined are considered absolute.

Example:

```
ADD AC2,X
LOC 200
ADD AC3,@Q2
LOC .+3,    SKIP 3 LOCATIONS
ADD AC1,AC2
```

RELOC - This is similar to LOC in that it explicitly sets the location counter. The block of code which follows is relocatable and all labels within the block are relocatable. The implicit statement:

```
RELOC 0;
```

begins all programs.

PHASE - A portion of code may be moved into other registers before it is executed. PHASE gives the location counter a value different from the location into which the assembled code is to be loaded. The

code is actually loaded into continuing sequential locations, but all labels within a PHASE'd area are in relation to the PHASE. Point elements (.) also relate to the PHASE. PHASE is followed by an expression indicating the first address of the sub-routine when it is to be executed.

```
        MOVE [XWD LOOPX,LOOP]
        BLT LOOP+5
         :
LOOPX:  PHASE 11
LOOP:   Z
        MOVN A(X)
        FMP MPYR
        FADM A(Y)
        SOJGE X, .-3
        JRST @LOOP
DEPHASE
```

This example is the central loop in a matrix inversion. Before executing it, the routine will be moved into fast accumulator memory locations 11 - 16. The symbol LOOP represents 11 and the point in the SOJGE instruction represents 15. The routine is, however, loaded into the normal sequential registers. A phased area is terminated by a DEPHASE, LOC or RELOC code. The DEPHASE code has no effect on the next sequential loading location, but restores the location counter to this value.

PASS 2 - This code causes the location assignment phase, PASS 1 to be suspended and PASS2 to commence.

END - This statement is the last statement in a program or subroutine. If it is a program, then the following expression is the location of the first instruction to be executed.

NOSYM - The assembler will normally output a symbol table or list of the symbols used and their definitions. The code NOSYM will suppress this.

ERRORS - The first couple of assemblies usually turn up a number of errors. This op-code will suppress all output except those items with errors, allowing convenient correction without unnecessary output.

LIT - This code will cause literals that have been previously defined to be assembled starting at the current location. If n literals have been defined, the next free cell will be at (.+n). This statement will have no effect on literals which are defined after it. LIT may not be used more than eight times.

VAR - This code will cause the symbols which have been defined by following them with (#) in previous statements to be assembled at this place as block statements. This has no effect on subsequent symbol definitions of this type. This, and the previous pseudo-op, LIT, are useful in controlling storage allocation. If these codes do not appear, all variables and literals are placed at the end of the program.

RIM - In paper tape assemblies, this code will cause binary output to be punched in RIM format.

OPDEF -

Define an operation mnemonic. This is followed by a symbol and a pair of brackets containing a statement that will generate one word of data. The symbol then becomes a mnemonic for the operation code represented by the 36 bit data word.

Example:

```
OPDEF PUSHP [PUSH PP,0]
OPDEF PUNCH [DATAO PTP,]
```

These opdefs may then be treated as ordinary op codes:

```
PUSHP X
PUNCH Y
```

SYN -

Define synonyms. This code is followed by 2 symbols or macros. The first must have been previously defined, and the second is made equivalent to it. If the first is a symbol, the second becomes a symbol with the same value; if the first is a macro, the second becomes a macro which acts identically; if the first is a machine-op, control code or data generating code, the second will behave in the same manner.

```
SYN K,X
SYN FAD,ADD
SYN END,XEND
```

If the first item is identical to both a symbol and a code, the second item (which is the synonym) is made synonymous with the symbol in preference to the code.

Listing Control

Several codes are used to control the final listing.

LIST -	Causes the processor to begin listing the assembled program, in both octal and source text.
XLIST -	Causes the processor to stop listing the assembled program.
LALL -	Causes the processor to list all text that is processed: macro expansions, list control codes, repeats, etc.
XALL -	Causes the processor to stop listing all text.
TITLE -	The comments field is written at the top of each printed page.
SUBTTL -	The comments field is written as a second line at the top of each printed page and the listing skips to the next page.
PAGE -	The listing begins a new page. (A form feed on the input tape also has this effect).

These list control codes are never printed in the final listings, except under LALL.

SECTION 4

RELOCATION AND LINKING

RELOCATION

The MACRO6 assembler will create a relocatable program. This program may be loaded into any part of memory and will presumably function properly. To accomplish this the address field of some instructions must have a relocation constant added to it. This relocation constant is equal to the difference between the memory location an instruction is actually loaded into, and the location it is assembled into. Most programs begin in location 60_8; if a program is loaded into cells beginning at location 1400_8, the relocation constant, K, would be 1320_8.

Obviously, not all instructions must be modified by the relocation constant. Consider the two instructions:

```
MOVEI 2, .-3
MOVEI 2, 1
```

The first is probably used in address manipulation and must be modified; the second probably should not. To properly accomplish the relocation, the actual expression forming an address is considered and modification is decided on this basis. Integer elements are "fixed" and not modified. Point elements are "relocatable" and are always modified.* Symbolic elements may be fixed or relocatable according to the means used in their definition. If a symbol is defined by direct assignment statement it may be relocatable or fixed depending on the expression following the =. If a symbol is defined as a macro, it is replaced by the string and the string itself must be considered. If it is defined as a label or a variable (#), it is relocatable.* Finally, references to literals are relocatable.*

To evaluate the relocatability of an expression, consider what happens at loading time. A constant, K, must be added to each relocatable element, and the resulting expression evaluated.

* Except under the LOC code which specifies absolute addressing.

Example:

A, B, C & D are relocatable
X = A + 2*B -3*C + D
k is the relocatability constant.
X_r = (A +k) + 2* (B+k) -3* (C + k) + (D + k)
X_r = A + 2*B -3*C+D+k

Thus, the expression X, under relocation, is relocatable.

Some conventions are followed concerning relocatability:

1) Only 2 values of relocatability for a complete expression are allowed, k and ∅.

2) An element may not be divided by a relocatable element.

3) Two relocatable elements may not be multiplied together.

4) Relocatable elements may not be combined by Boolean expressions.

If any of these rules are broken, the relocatability of the resulting expression is considered to be ∅, and the assembled code is flagged.

If a, c and b are relocatable symbols, then:

(a+b-c)	is relocatable
(a-c)	is fixed
(a+2)	is relocatable
(2*a)	is relocatable
(2#a-b)	is erroneous

LINKING SUBROUTINES

Programs usually consist of subroutines which must be linked. This is relatively easy if all of the subroutines are assembled together; they can be linked by means of JSR, SUBR instructions. If independently assembled, relocatable subroutines are used, linking them must be considered, since the symbol tables from the assembly are inaccessible to the loader.

To accomplish this linking, selected symbols are made available to the Linking Loader by the codes EXTERN and INTERN.

The EXTERN code identifies certain symbols as external to the program. The condensed object program contains the information that values for certain symbols must be derived from other programs at load time. An expression containing a reference to an external symbol must consist of only the single external symbol. The statement:

```
EXTERN SQRT, CUBE, TYPE;
```

identifies the symbols SQRT, CUBE and TYPE as external symbols. Symbols defined as external must not be defined as labels, variables, macros or assignments.

To make internal program symbols available to other subroutines as external symbols, the code INTERN is used. This code has no effect on the actual assembly of the subroutine, but will merely make a list of symbol equivalences available to other programs at loading time. The statement

```
INTERN SIN, COS, SIND, COSD;
```

might appear in a sin-cos routine where SIN, COS, SIND and COSD were entry points to the subroutine to calculate respectively sines and cosines of angles in radians and degrees. Internal symbols must be defined within the subroutine as assignments, labels or variables.

For example, if a square root is required, it would be called by

```
JSR, SQRT;
```

Elsewhere in the program would be the statement:

```
EXTERN SQRT;
```

In the square root subroutine would be the statement:

```
INTERN SQRT;
```

Some subroutines will have fairly common usage, and it will become convenient to place them in a library. To load these subroutines, the code LIBRARY is used. If the SQRT routine mentioned above were a library program, the statement:

```
LIBRARY SQRT;
```

would also appear in the program. The code LIBRARY is followed by a list of programs (expressed as 6 character identifying codes) required from the library.

```
LIBRARY KS 401, SQRT, ANFSQ2;
```

would cause the programs numbered KS401, SQRT and ANFSQ2 to be loaded from the library. Library routines each have their own internal symbols, and EXTERN statements are also necessary.

SECTION 5

MACRO INSTRUCTION

When writing a program, it is often desirable to abbreviate certain commonly used coding sequences. Consider the following example of coding which computes the length of a vector with components stored in 3 sequential locations:

Example:

```
MOVE Ø,V;        GET THE FIRST COMPONENT
FMP Ø;           SQUARE IT
MOVE 1,V&1
FMP 1,1
FAD 1;           ADD IN THE SQUARE OF THE SECOND
MOVE 1,V&2
FMP 1,1
FAD 1;           ADD IN THE SQUARE OF THE THIRD
JSR FSQRT;       USE THE FLOATING SQUARE ROOT ROUTINE
MOVEM LENGTH;    STORE THE LENGTH
```

A simpler method of coding would be to use a subroutine call:

```
JSP SJ,VLENGTH
EXP V,LENGTH
```

A macro instruction may be defined which will generate either of the above coding sequences with one statement:

```
VMAG VECT,LENGTH
```

This statement consists of a macro-name, followed by the arguments, in this case the location of the first component and the location of the memory cell in which to store the result. To be able to generate the above subroutine call with one statement, the programmer must define the macro VMAG. This is done by use of the DEFINE code.

```
DEFINE VMAG (A,B)
<JSP SJ, VLENGTH
EXP A,B>
```

This macro definition statement consists of the code DEFINE, followed by the name of the macro, a pair of parentheses containing a dummy string of arguments, and a pair of angle-brackets containing the coding to be generated each time the macro is called. A comment may appear between the parentheses and angle-brackets.

The string of dummy arguments is merely used as a model, and these may be any symbols that are convenient - usually single letters will do. The angle-brackets may contain any proper string of coding, normally, but are not restricted to a group of complete statements.

When the macro is called, real arguments are substituted for the dummy arguments in the definition. The coding in the angle-brackets is reproduced, except that each appearance of a dummy argument is replaced by the corresponding actual argument in the calling statement. In the example cited above, A and B are the dummy arguments in the definition. The calling statement has the real arguments VECT and LENGTH. The coding actually generated by the call is:

```
JSP SJ,VLENGTH
EXP VECT,LENGTH
```

The real arguments may be enclosed with parentheses or the parentheses may be omitted. If parentheses are used, the argument string is ended by a closed

parenthesis; if they are omitted, the argument string ends when all the arguments of the definition are filled, or when a carriage return, closed bracket, or semi-colon delimits an argument. When parentheses are used (this is implied by an open parenthesis following the macro name), all superfluous arguments are ignored. The above call for the vector length subroutine may have been written with equal validity as:

```
VMAG (VECT, LENGTH)
```

Arguments must be separated by commas. If an argument contains a comma, it must be enclosed by a pair of angle-brackets. These angle-brackets act only as argument delimiters and are stripped off in the actual argument. An example of this is:

```
DEFINE JEQ (A,B,C)
<MOVE (A)
CAMN B
JRST C>
```

If the data in Location B is equal to A, then the program jumps to C. If the contents of B must be compared to the instruction ADD 2,X; then the macro call would be written:

```
JEQ <ADD 2,X>, B, INSTX
```

Angle-brackets surrounding the ADD 2,X are removed and the proper coding will be generated.

A macro need not have arguments. The instruction:

```
DATAO PTP,PUNBUF(4)
```

which causes the contents of PUNBUF, indexed by register 4 to be output to the paper tape punch, may be generated by the macro PUNCH, defined by:

```
DEFINE PUNCH
<DATAO PTP, PUNBUF(4)>
```

This macro would be coded as:

```
PUNCH THE ANSWER
```

"THE ANSWER" becomes a comment when the macro is replaced by the defined pseudo code.

A macro need not be used in the statement code field. The string within the angle-brackets of the definition exactly replaces the macro name and its argument string. The macro:

```
DEFINE L(A,B)  <3*B-3*A+3>
```

gives an expression for the number of items in a table where 3 cells are used to store each item. A is the address of the first item and B is the address of the last item. To load an index register with the table length, one might write:

```
MOVE X,L(FIRST,LAST)
```

CREATED SYMBOLS

When a macro is called, it is often convenient to generate symbols, without explicitly stating them in the call. Created symbols accomplish this. Each time a macro that requires a created symbol is called, a symbol is generated and inserted in the macro. These generated symbols are of the form ..nnnn; that is, two decimal points followed by four digits. The first created symbol that is generated is ..ØØØ1, the next ..ØØØ2, etc.

If a symbol in a definition statement is preceded by a %, it is considered to be a created symbol. When a macro is called, all missing arguments that are of the form %X are replaced by created symbols, if they are so specified.

The following macro will cause the textual information "A" to be written out on the console, followed by a halt and a jump to B. The additional label is necessary since the text statement generates an indefinite number of data words. A created symbol is appropriate here since the programmer is probably not interested in the label.

```
DEFINE TYPE (A, %B)
<JSR TYPE
HALT %B
SIXBIT /'A'/
%B:>
```

This macro is called by:

```
TYPE HELLO
```

The call:

```
TYPE HELLO, BX
```

will not generate a created symbol. Instead, the explicit symbol overrides the created symbol.

CONCATENATION

The above example also illustrates the use of the concatenation operator, the apostrophe. An argument may not necessarily refer to a complete symbol but refers to a string of characters. The apostrophe may not be used otherwise within a macro definition, and further it is not a meaningful operator outside of a macro-definition. Another example is the macro:

```
DEFINE J(A,B,C)
<JUMP'A B,C>
```

The generated code when the macro is called by:

```
J LE,E,X&1
```

is:

```
JUMPLE 3,X&1
```

INDEFINITE REPEAT

Often in the definition of a macro, it is not known in advance how many arguments there will be. An example is a macro used to set up a symbol table. This table consists of a word of code corresponding to a 6 character symbol, followed by a word of temporary storage. There may be an indefinite number of symbols in the table. This is easily implemented by an indefinite repeat:

```
DEFINE STABLE(A)
<IRP (A)
<SIXBIT /'A'/
Z>>
```

The IRP A ⟨EXP⟩ indicates that A is composed of a string of sub-arguments separated by commas, and that the expression enclosed by angle-brackets is to be repeated with each sub-argument inserted in turn.
The above macro when called by:

```
STABLE<ALPHA, BETA, GAMMA;
```

generates:

```
SIXBIT /ALPHA/
Z
SIXBIT /BETA/
Z
SIXBIT /GAMMA/
Z
```

NESTING AND REDEFINITION

Macro definitions may appear within other definitions to any reasonable depth. A macro within another macro is not defined until the outer macro is called. The macro-processor simply substitutes the arguments into the defined string of characters, and nesting is wholly consistent with this type of operation.

If a macro name which has been previously defined appears within another definition statement, the macro is redefined, and the original definition is exorcised.

The first example, calculation of the length of a vector, may be used to illustrate this:

```
                    DEFINE VMAG (A,B,%C)
                    <JSP SJ,VL
                    EXP A,B
                    JRST %C

VL:                 MOVE 2,(SJ)
                    MOVE (2)
                    FMP Ø
                    MOVE 1,1(2)
                    FMP 1,1
                    FAD 1
                    MOVE 1,2(2)
                    FMP 1,1
                    FAD 1
                    JSR FSQRT
                    MOVEM @1(SJ)
                    JRST 2(SJ)
%C:
                    DEFINE VMAG (A,B)
                    <JSP SJ,VL
                    EXP A,B>>
```

This macro VMAG is defined as an entry to a closed subroutine followed by the closed subroutine. The nested redefinition redefines the macro as only the entry. The first time the macro is called, the subroutine is generated. Subsequent calls generate only the calling sequence - there is no need for another subroutine to be generated.

Another use of redefinition is a subroutine handler. On entering a subroutine, k accumulators are stored, and the prevailing radix is altered to one local to the subroutine. On exiting, the accumulators and radix are restored.

```
DEFINE EXIT <>
DEFINE ENTER  (R,K,%A,%B,%C,%D,%E)
<%B:     Z
         %A=10,        SAVE THE OLD RADIX
         RADIX R
REPEAT K <MOVEM .-%B+1, %C+.-%B+1> SAVE ACS
         SYN EXIT, %D;  SAVE DEFINITION
DEFINE EXIT  NEW DEFINITION
<%E:     RADIX %A,    RESTORE OLD RADIX
REPEAT K <MOVE .-%E, C+.-%E> RESTORE ACS
         SYN %D, EXIT; RESTORE DEFINITION
         PURGE %A,%D; REMOVE JUNK FROM TABLE
         JRST@%B,  RETURN FROM SUBROUTINE
%C:      BLOCK K,  AC STORAGE>    >
```

Each time ENTER is called, EXIT is redefined. To use this macro to store 4 accumulators on entering a subroutine and setting the local radix to 10, the following would be written:

```
SUBR:    ENTER (10,4)
```

ADDITIONAL CODES

CRSYM -

This code is analogous to VAR and LIT. It is a processor control code and indicates that previously undefined created symbols are to be given values according to the present contents of the location counter. Each undefined created symbol increments the location counter by 1.

SECTION 6

ERRORS

Occasionally, even the best of us commit small errors in writing programs. There are two classes of errors -- errors in language usage and program errors. MACRO6 will examine the statements for this first class of error, and print appropriate messages. These errors are caused by meaningless or inconsistent constructs in the source language. When a listing is prepared during the assembly, each MACRO6 statement that contains errors will be flagged by one or several letters in the margin. At the end of the listing will be a summary of the errors; this summary will be printed even if a listing is not prepared. Program errors which properly use the MACRO6 language will be correctly translated into errors in the binary program.

THE ERROR FLAGS

M Multiply defined symbol - a symbol is defined more than once, either as a label or variable. The symbol retains its original definition.

S Symbol Error - There is a meaningless character string that resembles a symbol or macro. It is assembled as though the value were zero.

P Phase Error - A symbol is assigned a value as a label during pass 2 different from that which it was assigned in pass 1. An error of this type will terminate an assembly; it would probably indicate an error in

conditional assembly or in macro redefinition and therefore propagate throughout the entire program. (Symbols re-assigned by "=" will not cause phase errors).

O — Undefined Code - The code indicating the statement type is not defined in the code table. It is assembled as a numeric code of zero.

N — Number Error - There is a meaningless string of characters that resembles a number. It is assembled as zero.

A — Argument Error - An argument of a control code has a peculiar value.

L — Literal - There is an error within a literal.

F — Macro Definition Error - A format error exists in a DEFINE statement.

U — Undefined Symbol - A symbol or macro is undefined. It is given a value of zero.

V — Value Previously Undefined - A symbol used to control the processor is undefined prior to the point at which it is first used.

R — Relocation Error - An expression has a relocation constant other than 1 or Ø, contains division by a relocatable number, contains the product of two relocatable numbers, or involves relocatable numbers in Boolean operations. The relocation constant is set to zero.

D Multiply Defined Symbol Reference - The statement contains a reference to a multiply defined symbol. It is assembled with the first value defined.

An error message in the summary will have the following format:

LOC	A + N	MACRO (n)	E
(Location Counter)	(Symbolic Address)	(Macro called depth)	(Error flags)

SECTION 7

ASSEMBLY OUTPUT

ASSEMBLY LISTING

There are two types of assembly output, the assembly listing and the binary program. The assembly listing consists of a printout of the source program. On the same line with each source statement are 3 numeric fields: the location of the assembled code, the left half word, and the right half word. Above each line containing an error is an appropriate message. This listing is controlled by the List Control Codes except that error messages are always printed. All assemblies begin with an implicit LIST.

BINARY PROGRAM

The binary program may assume two forms: RIM and LINK. The RIM (Read-in Mode) format is always punched into paper tape and is used for such things as loaders and computer hardware maintenance programs. RIM programs may be completely loaded by the loader resident in the shadow memory located "behind" the accumulator memory.

Rim Format

Programs in RIM mode consist of two word pairs. The first word is an instruction:

```
DATAI PTR, A,
```

The second word of the pair is the word of instruction or data to be loaded into memory location "A".

The last word of a RIM tape is a single instruction:

```
HALT, START;
```

where START is the first location of the program.

Link Format

LINK format is the normal binary output mode. Programs in this format are acceptable to the Linking Loader, and are generally relocatable. The Linking Loader will load sub-programs into memory, properly relocating each one and adjusting addresses to compensate for the relocation. It will also link External and Internal symbols to provide communication between independently assembled sub-programs. Finally, the Linking Loader will call and load library sub-routines.

LINK format data is in blocks. All blocks have an identical format. The first word of a LINK block consists of two halves. The left half is a code for the block type, and the right half is a count of the number of data words in the block. The data words are grouped in sub-blocks of 18 items. Each 18 word sub-block is preceded by a relocation word. This relocation word consists of 18 two bit bytes. Each byte corresponds to one word in the sub-block, and contains relocation information regarding that word.

If the byte value is:
- 0 - no relocate occurs
- 1 - the right half is relocated
- 2 - the left half is relocated
- 3 - both halves are relocated

These relocation words are not included in the count; they always appear before each sub-block of 18 words or less to insure proper relocation.

This block format is universal. All programs (except those in paper tape RIM format) are stored in this format, including programs on paper tape, Microtape, standard magnetic tape, punched cards, drums and discs. This format is totally independent of logical divisions in the input medium (40 word check summed paper tape blocks, 128 word blocks on Microtape and drums, 23 word check summed punched cards, etc). It is also independent of the block type.

The Formats for the Block Types

Code 1 - Program and Data (assembler and compiler output)

Data word 1 - The location of the first word in the block

Data words 2-N - Up to N-1 words of program or data to be loaded.

Code 2 - Program symbol table (local symbols)

The data words are in pairs, the first is the symbol (in 6 bit ASCII) and the second is the value. This block is necessary for debugging routines.

Code 3 - Internal/external symbol table

The data words are in pairs.

Data word 1 - symbol (6 bit ASCII)

Data word 2 (LH) - 1 for a right hand external
2 for a left hand external
4 for an internal

(RH) - symbol value for internal

\- link for external

These symbols are used to link independently assembled subroutines. The second and subsequent occurrence of an internal symbol is ignored.

Code 4 - Library requests.

Each data word is the name (6 bit ASCII) of a library routine to be loaded.

Code 5 - Highest relocatable point.

This is the last block in a subroutine. It contains 1 word, which is the location of the highest memory address used by the relocatable program. Upon loading, the relocation constant (initially set to zero) is replaced by this number to properly relocate the next subprogram.

Code 6 - Name

There is 1 data word with the subroutine name. This usually is the first block.

Code 7 - Starting address

There is 1 data word with the starting address of the program. This block should only occur in conjunction with the main program. The second and subsequent occurrences of this block are ignored.

Code 10 - Combined internal-external

The data words are split into a right half and a left half. The right half is the link, and the left half the value. These items are used to control forward references in one pass compilers.

SECTION 8

PROCESSOR INITIALIZATION

At the beginning of each assembly the assembler is initialized to certain states, generally affected by control codes. The initial states are:

1. Radix is set to 8.
2. The location counter is set to zero and relocatable assembly will occur.
3. There will be a normal listing.
4. There will be LINK binary output with a symbol table.
5. Phase mode is off.
6. The title and subtitle are blanked.
7. Only device mnemonics are placed in the symbol table. They are:

APR	= ØØØ	Arithmetic Processor
PI	= ØØ4	Priority Interrupt System
PTP	= 1ØØ	Paper Tape Punch
PTR	= 1Ø4	Paper Tape Reader
CP	= 11Ø	Card Punch
CR	= 114	Card Reader
TTY	= 12Ø	Console Teleprinter
LP	= 124	Line Printer
DIS	= 13Ø	Display
DC	= 2ØØ	Data Control
UTC	= 21Ø	Micro Tape Control
UTS	= 214	Micro Tape Status
MTC	= 22Ø	Mag Tape Control
MTS	= 224	Mag Tape Status
MTM	= 23Ø	Mag Tape Status
DCSA	= 3ØØ	Data Communication System
DCSB	= 3Ø4	Data Communication System

8. No Macros or Opdefs exist.

APPENDIX I

CODES

Data Generating Codes

DEC - Decimal Numbers

OCT - Octal Numbers

EXP - Expressions

XWD - Block Transfer Word

IOWD - Input/Output Transfer Word

POINT - Pointer Word

SIXBIT - ASCII (6 bit) character strings

BYTE - Variable length bytes

BLOCK - Block of storage reserved

ASCII - ASCII (7 bit) character strings

Processor Control Codes

REPEAT - Repeat character string

IFn - Conditional Assembly

n	Condition
E	zero
G	positive
GE	zero or positive
L	negative
LE	zero or negative
N	non zero
B	blank
1	pass 1
2	pass 2

OPDEF - Define an op mnemonic

SYM - Define a synonym

PHASE - Enter Phase mode

DEPHASE - Leave Phase mode

RIM - Assemble RIM tapes

IFIDN - Conditional Assembly on character strings

IFDIF - Conditional Assembly on character strings

RADIX - Radix control

LOC - Set Location Counter

PASS2 - Terminate Pass 1

NOSYM - Suppress Symbol Table Output

LIT - Assemble Literals

VAR - Assemble Variables

CRSYM - Assemble Created Symbols

EXTERN - List of External Symbols

INTERN - List of Internal Symbols

LIBRAR - List of Library Subroutines

IRP - Indefinite Repeat

PURGE - Purge Symbols

END - Last Line

List Control

LIST - List

XLIST - Stop Listing

LMAC - List Macro Expansions

XMAC - Stop Listing Macro Expansions

TITLE - Title

SUBTTL - Subtitle

PAGE - Skip to top of next page

ERRORS - Suppress Output except for error messages

APPENDIX II

SUMMARY OF ERROR FLAGS

A -	Argument of Control Op
D -	Reference to multiply defined symbol
F -	Macro Definition
L -	Useage of Literal
M -	Multiply defined symbol
N -	Number
O -	Undefined Operation Code
P -	Phase Discrepancy
R -	Relocation
S -	Symbol
U -	Undefined Symbol
V -	Value previously undefined

APPENDIX III

PROGRAMMING EXAMPLE

```
TITLE BUG6

,               THIS IS A GAME IN WHICH A SMALL CIRCLE RUNS AROUND THE
,               FACE OF THE SCOPE.  IT STARTS RUNNING AT A HIGH SPEED
,               AND EVENTUALLY SLOWS DOWN, STOPPING ON ONE OF THE 12
,               CLOCK POSITIONS.  MOMENTARILY FLIPPING DATA SWITCH 35
,               UP WILL RESTART THE BUG AND IT WILL REPEAT THE CYCLE,
,               STOPPING ON A NEW POSITION.  RANDOM NUMBER GENERATOR
,               IS USED TO DETERMINE THE AMOUNT OF TIME THE BUG WILL
,               DWELL AT ANY POSITION.  THIS RANDOM NUMBER IS INITIA-
,               LIZED FROM THE SWITCHES.


BEGIN:          HALT .+1;       STOP TO SET THE DATA SWITCHES
                                ,INITIALIZE THE FOUR CELLS OF THE
                                ,RANDOM NUMBER GENERATOR TO Ø, A
                                ,NUMBER READ FROM THE SWITCHES, ANOTHER
                                ,Ø, AND A CONSTANT.
                RSW R2
                CLEARB R1,R3
                MOVE R4,[123456701231
                MOVEI CLOCK,Ø; START AT TWELVE OCLOCK
                JSR SCOPE;      START THE SCOPE RUNNING

, RUN THE SPOT AROUND THE SCOPE

GO:             CLEARM TIMER;                   FIRST ZERO THE ACCUMULATING

RUN:            JSR RAND;                       TIMER, GET A RANDOM NUMBER
                ANDI DELTAT,DELMAX;             LIMIT THE RANDOM NUMBER
                ADD TIMER,DELTAT;               ADD IT TO THE ACCUMULATING
                MOVE WAIT,TIMER;                TIMER, MOVE THE TOTAL TO A
                                                ,COUNTER

                SOJN WAIT,.;                    COUNT DOWN TO KILL TIME

                MOVE POSITION(CLOCK)            ADVANCE CLOCK TO NEXT POSIT.
                MOVEM TABLE+1
                SOJGE CLOCK,.+2;                HAVE WE PASSED TWELVE OCLOCK
                MOVEI CLOCK,↑D11;               RESET CLOCK BACK TO ONE

                TDNN TIMER, TMAX;               ARE WE READY TO STOP IT
                JRST RUN;                       NO
                RSW TIMER;                      YES, BUT INSTEAD OF ACTUALLY
                TRNN TIMER,1;                   HALTING, IT LOOPS UNTIL DS35
                JRST .-2;                       IS SET. THIS ALLOWS THE
                JRST GO;                        DISPLAY TO CONTINUE
DELMAX=7777,                                    MAXIMUM TIME INCREMENT
TMAX:           777700000000,                   MAXIMUM TOTAL TIME
```

```
,RANDOM NUMBER GENERATOR

,               THIS RANDOM NUMBER GENERATOR ADDS FOUR RANDOM NUMBERS
,               MODULO 2↑35 TO GET A FIFTH RANDOM NUMBER.  THEN IT
,               REPLACES THE FIRST BY THE SECOND, THE SECOND BY THE
,               THIRD, THE THIRD BY THE FOURTH, AND THE FOURTH BY THE
,               FIFTH TO RESET THE GENERATOR.  THE FOURTH IS THE GENE-
,               RATED RANDOM NUMBER.

RAND:           Z
                MOVE DELTAT,R1;                 ADD FOUR RANDOM NUMBERS
                ADD DELTAT,R2
                ADD DELTAT,R3
                ADD DELTAT,R4

                MOVE R1,R2;                     REPLACE VALUES
                MOVE R2,R3
                MOVE R3,R4
                MOVE R4,DELTAT

                JRST @RAND
```

```
,DISPLAY ROUTINE
SCOPE:          Z
                CONO PI,10400;          SHUT DOWN THE PI SYSTEM
                EXCH TBLPT1;            INITALIZE THE TABLE POINTER
                MOVEM TBLPTR#;          WITHOUT DISTURBING AC0
                EXCH TBLPT1;
                CONO DIS,112;           START THE DISPLAY ASSIGNING
                                        ,SPECIAL CHANNEL 1 AND DATA
                                        ,CHANNEL 2
                DATAO DIS,TABLE;        GIVE THE SCOPE ITS FIRST
                                        ,WORD
                CONO PI,2340;           TURN ON THE PI SYSTEM, ACT-
                                        ,IVATING CHANNELS 1 AND 2.
                JRST @SCOPE;

,DISPLAY MACROS

  DEFINE INCREMENT (A,B,C,D,E,F,G,H,I)
    A WORD OF SCOPE INCREMENTS
    <BYTE (2)A(4)B,C,D,E(2)A(4)F,G,H,I>

,SOME DEFINITIONS TO USE IN THE INCREMENT MACRO

                RADIX2
I=1,                            DISPLAY THE POINTS (INTENSIFY
ESCAPE=10,                      EXCAPE FROM THE MODE
PX=1000,                        INCREMENT:      PLUS X
PXPY=1010,                                      PLUS X AND PLUS Y
PY=0010,                                        PLUS Y
MXPY=1110,                                      MINUS X AND PLUS Y
MX=1100,                                        MINUS X
MXMY=1111,                                      MINUS X AND MINUS Y
MY=0011,                                        MINUS Y
PXMY=1011,                                      PLUS X AND MINUS Y
                RADIX8

DEFINE XYPOS (A,B)      A WORD TO POSITION THE DISPLAY
                AND THEN START THE INCREMENT MODE

                <BYTE (2)0(3)1,1(10)A(2)1(3)6,1(10)B;>
```

```
,DISPLAY TABLE

TBLPT1:         IOWD Ø,TABLE;                   INITIAL IO WORD FOR SCOPE
                                                ,TABLE
TABLE:          BYTE (18)(5)1(4)(3)1(2)2(1)1(3)5;           START IN XY
,TRACE A CIRCLE IN THE INCREMENT MODE
                XYPOS 1ØØØ,16ØØ;                            MODE
                INCREMENT I,PX,PX,PXPY,PX,PXPY,PX,PXPY,PXPY
                INCREMENT I,PY,PXPY,PY,PXPY,PY,PY,PY,PY
                INCREMENT I,MXPY,PY,MXPY,PY,MXPY,MXPY,MX,MXPY
                INCREMENT I,MX,MXPY,MX,MX,MX,MX,MXMY,MX
                INCREMENT I,MXMY,MX,MXMY,MXMY,MY,MXMY,MY,MXMY
                INCREMENT I,MY,MY,MY,MY,PXMY,MY,PXMY,MY
                INCREMENT I,PXMY,PXMY,PX,PXMY,PX,PXMY,PX,PX

                BYTE (1)1(17)(7)(1)1(1)1;       EXCAPE FROM INCREMENT MODE
                                                ,AND STOP

,CLOCK POSITIONS
POSITION:       XYPOS 1ØØØ,16ØØ;                TWELVE OCLOCK
                XYPOS 5ØØ,152Ø;                 ELEVEN
                XYPOS 26Ø,13ØØ;                 TEN
                XYPOS 2ØØ,1ØØØ;                 NINE
                XYPOS 26Ø,5ØØ;                  EIGHT
                XYPOS 5ØØ,26Ø;                  SEVEN
                XYPOS 1ØØØ,2ØØ;                 SIX
                XYPOS 13ØØ,26Ø;                 FIVE
                XYPOS 152Ø,5ØØ;                 FOUR
                XYPOS 16ØØ,1ØØØ;                THREE
                XYPOS 152Ø,13ØØ;                TWO
                XYPOS 13ØØ,152Ø;                ONE

,INTERRUPT TRAPS

LOC 42
                JSR SCOPE;      CHANNEL ONE FOR SCOPE STOP
                Z
                BLKO DIS,TBLPTR;                CHANNEL TWO  FOR DATA INTE

,ACCUMULATOR ASSIGNMENTS
R1=7
R2=6
R3=5
R4=4
DELTAT=3
TIMER=2
CLOCK=1


END BEGIN
```

APPENDIX IV

CHARACTER SETS

	ASCII	6 bit ASCII	Punched Card
(space)	240	00	b
!	241	01	12-7-8
"	242	02	0-5-8
#	243	03	0-6-8
$	244	04	11-3-8
%	245	05	0-7-8
&	246	06	11-7-8
'	247	07	6-8
(	250	10	0-4-8
)	251	11	12-4-8
*	252	12	11-4-8
+	253	13	12
,	254	14	0-3-8
-	255	15	11
.	256	16	12-3-8
/	257	17	0-1
ø	260	20	ø
1	261	21	1
2	262	22	2
3	263	23	3
4	264	24	4
5	265	25	5
6	266	26	6
7	267	27	7
8	270	30	8
9	271	31	9
:	272	32	11-0
;	273	33	0-2-8
<	274	34	12-6-8
=	275	35	3-8
>	276	36	11-6-8
?	277	37	12-0

	ASCII	6 bit ASCII	Punched Card
@	300	40	4-8
A	301	41	12-1
B	302	42	12-2
C	303	43	12-3
D	304	44	12-4
E	305	45	12-5
F	306	46	12-6
G	307	47	12-7
H	310	50	12-8
I	311	51	12-9
J	312	52	11-1
K	313	53	11-2
L	314	54	11-3
M	315	55	11-4
N	316	56	11-5
O	317	57	11-6
P	320	60	11-7
Q	321	61	11-8
R	322	62	11-9
S	323	63	0-2
T	324	64	0-3
U	325	65	0-4
V	326	66	0-5
W	327	67	0-6
X	330	70	0-7
Y	331	71	0-8
Z	332	72	0-9
[	333	73	11-5-8
\	334	74	7-8
]	335	75	12-5-8
↑	336	76	5-8
←	337	77	7-9