**DATA GENERAL
CORPORATION**
Southboro,
Massachusetts 01772
(617) 485-9100

PROGRAM

Extended Assembler

TAPES

Absolute Binary: 091-000017

ABSTRACT

The extended assembler, like the basic assembler, converts symbolic assembly
statements into machine language code. In addition to basic assembler features
the extended assembler provides relocation, interprogram communication, condi-
tional assembly and more powerful number definition facilities.

093-000040-00

EXTENDED ASSEMBLER

The extended assembler differs from the basic assembler in four respects:
1) *Relocatability* - programs can be assembled so that they may be loaded by the relocatable loader; 2) *Interprogram communication* - programs can be assembled which reference data, instructions, and addresses defined in other programs or vice-versa (argument swapping or sharing); 3) *Number definition* - simpler methods for specifying double precision, decimal and floating point constants as well as bit boundary alignment of constants are provided; 4) *Conditional assembly* - whole programs or portions of programs can be assembled or by-passed on the basis of an absolute expression evaluating to zero.

Except for these added features the extended assembler is identical to and compatible with the basic assembler, and a knowledge of the basic assembler (see write-up 093-000017) is a prerequisite in the following discussion. The extended assembler is also compatible with the basic to the extent that programs not using the relocatable or interprogram communication facilities (*ie* with no occurrence of one of the pseudo-ops: .ZREL, .NREL, .TITL, .ENT, .EXTN, or .EXTD) will be assembled as absolute and the binary tape will be punched in the same format as the output of the basic assembler for loading by the absolute binary loader.

Use of the relocation and interprogram communication facilities requires deferment of final address assignment until load time thus leaving this task to the relocatable loader. It is not surprising then that the relocatable loader (see write-up 093-000039) is a more sophisticated program than the absolute binary loader and that the data passed to it by the extended assembler differs from the output of the basic assembler passed to the absolute binary loader.

Together the extended assembler and relocatable loader provide a package that enables the programmer to work separately on subprograms in the coding,

debugging and testing phases without worrying about the absolute location of
a given program or the absolute locations of data and addresses shared by
programs at run-time.

RELOCATION

In addition to the assembly of absolute code the user may use the extended
assembler to produce two types of relocatable code. These types will be referred
to as *zero* relocatable and *normal* relocatable.

Storage words that may be relocated but must reside in page zero should be
assembled as zero relocatable using the zero relocatable mode. The user informs
the assembler that a body of code is to be zero relocatable by preceding it with
the pseudo-op .ZREL. Likewise, storage words that may be relocated anywhere
except page zero must be assembled as normal relocatable in normal relocatable
mode. The user indicates this by preceding his normally relocatable code with
the pseudo-op .NREL.

The extended assembler initially assumes assembly mode to be absolute and
continues to assemble in this mode until it encounters an occurrence of either
.ZREL or .NREL in the user's code. The user may enter zero relocatable or
normal relocatable mode at any point in his program simply by issuing a .ZREL
or .NREL pseudo-op and the assembler will pick up the assembly at the next
unused zero relocatable or normal relocatable address. Also, having once entered
one of the relocatable modes with a .ZREL or .NREL pseudo-op and having defined
symbols in that mode, the .LOC pseudo-op with an expression containing a
previously defined symbol or symbols may be used to reenter that mode. The type
of the expression determines the mode entered. Thus, when the expression used
in the pseudo-op evaluates to a zero relocatable value the zero relocatable
mode is entered and the zero relocatable relative location counter is set to
the next unused zero relocatable address or to the value of the expression if

it is higher. Likewise, when the expression evaluates as either normal reloca-
table or absolute, the normal relocatable relative location counter or the
absolute location counter is set. At no time, however, may the .LOC pseudo-op
be used to move either of the two relative location counters or the absolute
location counter backwards, since this would create the possibility of overwriting
portions of the preceding code which is not permitted by the relocatable loader.
The . used in an expression associated with the .LOC pseudo-op has the meanings:
"current absolute address", "current normal relative address", and "current page
zero relative address" when used within absolute, normal relocatable, and page
zero relocatable code respectively. The following statements provide examples
of the use of these pseudo-ops.

```
- - -
       00000 000000              0               ;ABSOLUTE
       00001 000000              0
             000027              .LOC 27         ;ADJUST ABS LOC COUNTER
       00027 000170    A:        2*TABL
       00030 000113    B:        TABL+17
             000074              .LOC .+43       ;ADJUST ABS LOC COUNTER
             000020    TABL:     .BLK 20

                                 .ZREL           ;ZERO RELOCATABLE
       00000-003510    PNTR:     SUBRT
       00001-000000'   PNTR1:    MAIN
             000007-             .LOC .+5        ;ADJUST ZREL LOC COUNTER
       00007-000000    ARG1:     0

             000377              .LOC ARG1-PNTR+370      ;ABSOLUTE
       00377 000000    ARG2:     0

                                 .NREL           ;NORMAL RELOCATABLE
       00000'022027    MAIN:     LDA 0,#A
       00001'024030              LDA 1,B

             000010-             .LOC ARG1+1     ;ZERO RELOCATABLE
       00010-010074              ISZ TABL
L                                .LOC PNTR1+2    ;LOC COUNTER CAN'T GO BACK
       00011-020006              LDA 0,ARG1-PNTR1

             003510              .LOC 3510       ;ABSOLUTE AGAIN
       03510 024007-   SUBRT:    LDA 1,ARG1
       03511 030377              LDA 2,ARG2
L                                .LOC 3500       ;LOC COUNTER CAN'T GO BACK
       03512 010400              ISZ SUBRT+2

             000006'             .LOC MAIN+6     ;NORMAL RELOCATABLE
       00006'052027              STA 2,#A

                                 .END
```

The assembler will begin assembly in absolute mode and will create two data words containing zeros for locations 00000 and 00001, then the .LOC pseudo-op with the absolute expression 27 will change the location counter to 27_8 and the next two data statements will create two data words which at run time will contain respectively a *byte pointer* to the table called TABL, and the address of the end of the table. The next two statements increment the location counter 43_8 times and reserve 20_8 locations to store the table. The .ZREL pseudo-op causes the assembler to shift from absolute to zero-relocatable mode, and the next two statements, which will be assigned zero relocatable relative addresses 00000 and 00001, reserve words which are loaded with the relative addresses of routines SUBRT and MAIN. ARG1 is assigned zero relocatable relative address 00007 since the .+5 expression used in the .LOC pseudo-op increments the zero relocatable location counter by five. ARG2 is assigned absolute address 377_8 since the expression ARG1-PNTR + 370 from the preceding .LOC pseudo-op evaluates to an absolute value thus shifting the assembler into absolute mode. The .NREL then shifts the assembler into normal relocatable mode and proceeds to assemble the routine MAIN starting at the normal relocatable relative address 00000. But, because when evaluated the expression ARG1+1 in the next .LOC pseudo-op is zero relocatable, the assembler then returns to zero relocatable mode. Note that the instruction ISZ TABL which follows this .LOC is assigned page zero relocatable address 10_8 which is both the value of the expression ARG1+1 and the next available unused page zero relocatable address. Hence, exactly the same result could have been obtained by replacing the .LOC ARG1+1 statement with a simpler .ZREL statement. In practice this would be the most usual way of assigning the subsequent statement the next available relocatable address. The expression PNTR1+2 in the next .LOC pseudo-op is zero relocatable hence the assembly mode remains unchanged, but because the expression evaluates to page zero relocatable address 00003 which has already been used the

pseudo-op receives an L flag and the subsequent statement is assigned page zero relocatable address 00011 which is the next available page zero relative location. The .LOC 3510 shifts the assembler back to absolute mode, but the .LOC 3500 although keeping the assembler in absolute mode is given an L flag since in order to complete the command the assembler would have to turn back the absolute location counter.

Expression Evaluation

The extended assembler allows expressions using relocatable symbols, but certain restrictions should be kept in mind when constructing them:
1) Expressions using both page zero and normal relocatable symbols must be such that either the page zero or the normal symbols cancel out. Thus for example, expressions of the form $Z_1 + N_1 - Z_2 + N_2 - N_3$ are legal, where the Zs represent page zero relocatable symbols and the Ns normally relocatable symbols.
2) Expressions that evaluate to twice a relocatable symbol or the sum of two like relocatable symbols are permitted in data statements but those that will evaluate to higher multiples or non-integer multiples of relocatable symbols are illegal. 3) Externally defined symbols (relocatable or absolute), op codes, double precision, and floating point numbers are all unuseable in expressions.

The last point is straightforward, but the first two require consideration of the loading process to be understood. During loading the loader must add a constant K_1 to each N_i and a constant K_2 to each Z_i in the program it is loading to determine the absolute addresses of the symbols in the loaded program. It also must modify the contents of each storage word appropriately which it does by adding one and only one of five possible constants to the word $(0, K_1, K_2, 2K_1,$ or $2K_2)$. From this one can see that expressions that mix page zero and normal relocatable symbols without one or the other cancelling will not be allowed. Also one can see that loader modification of address contents by more than twice a relocatable base is not permitted. At this point it is important to see why

expressions evaluating to twice a relocatable symbol *are* permitted. Probably

the most common use for expressions of this kind is in the creation of byte

pointers in data statements for use by input/output routines that process 8 bit

bytes. This is discussed on page 2-21 of How to Use the Nova, but breifly, a

byte pointer is a storage word in which bits 0-14 contains an address and bit 15

specifies which half of the word addressed is to be worked on. Clearly, a byte

pointer of this kind can be formed by simply doubling an address, and can be

retrieved and regenerated by a simple shifting operation. It should be

remembered that this is a convenient software convention and is not a hardware

function. We shall use the terms *byte pointer type relocatable* or *byte*

relocatable to describe expressions of this kind.

It is also important to stress here that these byte pointer type relocatable

expressions are only permissable in data statements and are not acceptable as

addresses in memory reference instructions. Expressions used in the address

portion of memory reference instructions must evaluate to be absolute, page zero

relocatable or normal relocatable.

Specifically, expressions of the following forms or which can be reduced

to these forms are acceptable to the extended assembler and produce values having

the properties stated.

Expression	Attribute of Evaluated Result
A±A	Absolute
R-R	Absolute
R±A	Relocatable
R+R	Byte Relocatable
2*R	Byte Relocatable

Expressions that cannot be evaluated to be absolute, relocatable, or byte

pointer type relocatable, as well as those that illegally mix page zero and

normal relocatable symbols will receive R error flags.

As part of the assembly listing the extended assembler prints the address assigned (absolute or relative) and the contents (before load time) of each storage word generated by the assembler from the programmer's source code. In the listings it flags each address to indicate in what way that storage word may or may not be relocated, and flags the contents of the address to indicate how they will be affected in relocation. These flags will be printed adjacent to and to the right of these octal fields on the listing. The flags are:

Address Flags	Meaning
blank	Address of word is absolute.
-	Address of word is page zero relocatable.
'	Address of word is normally relocatable.

Content's Flags	Meaning
blank	Contents of word are absolute.
-	Contents of word are page zero relocatable.
=	Contents of word are page zero byte relocatable.
'	Contents of word are normally relocatable.
"	Contents of word are normally byte relocatable.
$	Storage word references a displacement external.

These flags can be seen in the previous example and in the following.

```
---
                                .EXTD  DISP

        000010                  .LOC 10
 00010 000006     A:           6
 00011 000000                  0


        000001-                 .ZREL
                                .LOC .+1
 00001-000400     B:           400                     ;CONTENTS ABSOLUTE
 00002-000001-    C:           B                       ;CONTENTS PZ RELOCATABLE
 00003-000004=    D:           C+C                     ;CONTENTS PZ BYTE REL.
 00004-000005-                 JMP .+1
 00005-034002-                 LDA 3,C
 00006-031525                  LDA 2,125,3
A 00007-044000                 STA 1,G                 ;ADDRESS OUTSIDE PAGE ZERO
 00010-000001'    W:           F                       ;CONTENTS NORM RELOCATABLE
 00011-000002"    X:           F+F                     ;CONTENTS NORM BYTE REL.
 00012-020001$                 LDA 0,DISP              ;ADDR IS DISPLACEMENT EXTERNAL


                                .NREL
 00000'000010     E:           A                       ;CONTENTS ABSOLUTE
 00001'000000'    F:           E                       ;CONTENTS NORM RELOCATABLE
 00002'000000"    G:           E+E                     ;CONTENTS NORM BYTE REL.
 00003'000401                  JMP .+1
 00004'034775                  LDA 3,F
 00005'031525                  LDA 2,125,3
 00006'044003-                 STA 1,D
R 00007'000000    H:           E+E+E    ;EXPRESSION NOT ABS, REL, OR BYTE REL.
R 00010'000000    I:           5*F      ;DITTO
R 00011'000000                 5*F/2    ;DITTO
R 00012'000000                 4*F/3    ;DITTO
R 00013'000000                 2*F/3    ;DITTO
R 00014'000000    J:           C+F      ;UNCANCELLED MIX OF PZ & N REL SYMBOLS
A 00015'020000                 LDA 0,2*Y+G-E-Z ;ERROR - .-200>ADDRESS>.+177
        000416'                 .LOC .+400
 00416'020402                  LDA 0,2*Y+G-E-Z ;O.K. - .-200<ADDRESS<.+177
 00417'000002-    Y:           C                       ;CONTENTS PZ RELOCATABLE
 00420'000004=    Z:           C+C                     ;CONTENTS PZ BYTE REL.
 00421'040001$                 STA 0,DISP              ;ADDR IS DISPLACEMENT EXTERNAL

        000013-                 .LOC W+3                         ;ZERO RELOCATABLE
A 00013-010000                 ISZ 3*W+D/2             ;ERROR - ADDR IS PZ BYTE REL.
 00014-010006-                 ISZ 3*W+D/4             ;O.K. - ADDR IS PZ RELOCATABLE
 00015-000006-                 3*W+D/4                 ;CONTENTS PZ BYTE REL.

        000427'                 .LOC Z+7                         ;NORMAL RELOCATABLE
A 00427'014000                 DSZ 4*Y-F-Z+C-D ;ERROR - ADDR IS NORM BYTE REL.
 00430'014575                  DSZ 4*Y-F-Z+C-D/2       ;O.K. - ADDR IS NORM REL
 00431'001452"                 4*Y-F-Z+C-D             ;CANCELLED MIX OF PZ & N REL SYM
Z 00432'034000                 LDA 3,DISP+6            ;EXTERNAL USED IN EXPRESSION
Z 00433'010000                 ISZ JMP+4               ;OP CODE USED IN EXPRESSION
Z 00434'050000                 STA 2,6D+3              ;DOUB PREC # USED IN EXPR.
Z 00435'000000                 2.0*Z                   ;FLTG PNT # USED IN EXPR.
C                 JMP:         0                       ;OP CODE USED AS SYMBOL

                                .END
```

The example above also shows three types of error flags: the 'R', the 'A', and the 'Z'. The 'R' flag, as has been mentioned before, is used to flag expressions that cannot be evaluated to be absolute, relocatable or byte pointer type relocatable, or which mix page zero and normal relocatable symbols in a non-cancelling fashion.

The 'A' flag plays much the same role as it does in the absolute assembler indicating address errors. That is, when *memory reference* instructions (JMP, JSR, ISZ, DSZ, LDA, & STA) that are to be page zero relocatable reference addresses outside page zero, or when those that are to be normally relocatable reference addresses outside the range of location counter relative addressing (.-200 $\leq$ address $\leq$.+177), or when an expression used to specify an address does not evaluate to an acceptable absolute, page zero relocatable, or normal relocatable address, the statements will be flagged with an 'A' and an absolute address of 00000 will be substituted by the assembler. Expressions used in *data* statements are not restricted in the addresses they reference and hence when assembled may contain byte relocatable as well as absolute or ordinary relocatable data.

The 'Z' flag is generated whenever a statement contains an expression that uses symbols not evaluable by the assembler. These expressions containing externals, op codes, double precision numbers, and floating point numbers will receive 'Z' flags.

INTERPROGRAM COMMUNICATION

It is possible using the extended assembler to reference data, addresses and constants in a program which are not defined within that program but rather in others, and it is also possible to make symbols defined within that program available to other programs by preceding the program code with pseudo-ops declaring the appropriate symbols as either externals or entries. Note that although a symbol may be used in many programs to reference some datum, address

or constant, it can only be defined in one program without being multiply-defined. Hence, within a suite of programs a symbol may be declared as an external by several programs but should be declared as an entry in only one program.

Two types of externals may be specified using the extended assembler. They will be referred to as *normal externals* and *displacement externals* (or *external displacements*). Displacement externals may be used in any memory reference instruction or data statement, but when evaluated by the assembler must resolve to a value representable in eight binary digits. That is, when used in a data statement or in a memory reference instruction with index = 00 (referring to page zero) the displacement must resolve to a value in the range $0 \leq D \leq 377$; when used in a memory reference instruction with index $\neq$ 00 (addressing relative to the location counter or relative to a base address contained in AC2 or AC3) the displacement must fall in the range of permissable displacements $-200 \leq D \leq 177$. Normal externals are permissable only in data statements, *ie*, an entire storage word must be reserved for a normal external.

Two pseudo-ops are used to declare symbols that will be used as normal externals or displacement externals. The pseudo-op used to declare normal externals has the form

$$.\text{EXTN} \quad \text{S1, S2 ...}$$

where S1, S2 represent the symbolic names of the normal externals. These symbols must conform with the rules for symbol definition applicable to all other symbols. At least one symbol must be specified, but any number may appear if separated by spaces or commas. The pseudo-op for declaring displacement externals has the same form

$$.\text{EXTD} \quad \text{S1, S2 ...}$$

where S1, S2 represent the names of the displacement externals. Every external must be declared in some other program as an entry by means of the entry pseudo-op which has the form

$$.\text{ENT} \quad \text{S1, S2 ...}$$

where S1, S2 represent symbols defined within the current program. Programs
which use externals or define entries must declare the relevant symbols in the
.ENT, .EXTN, OR .EXTD pseudo-ops before any other statements. The order, how-
ever, in which these three pseudo-ops appear is immaterial. Any errors that
occur in the declaration of internal or external symbols are indicated by
flagging the statement with a G flag.

Since titles are key identifying elements required by the symbolic debugger
and library file editor used in conjunction with the extended assembler, a
pseudo-op for naming programs is provided. This statement takes the form

.TITL *title*

where *title* represents a legitimate symbol which becomes the program name. This
symbol may conflict with any other symbol without causing an error, since this
symbol is implicitly different from all others. However, if the title violates
the rules for symbol definition, the statement will receive a G flag. If no
.TITL statement is included in a program, the assembler assumes the title .MAIN,
and this will be the symbol punched in the title block (see Appendix A). The
.TITL statement must appear before any statement that generates object data.
If a second .TITL appears before a data statement, the first title is replaced
by the second.

The example that follows illustrates use of the external, entry, and title
facilities.

```
                              .TITL    REPUS

                              .ENT     BGN,CCRLF,.CRLF
                              .EXTN    CRLF,TYPET
                              .EXTD    C377,DONE

                              .ZREL
  00000-001764"  CSTR:        STRING+STRING
  00001-001776"  CSTR1:       STRING+STRING+12
         000001  PNTR:        .BLK 1
  00003-177777   .CRLF:       CRLF
  00004-005015   CCRLF:       5015


                              .NREL
  00000'006003-  BGN:         JSR @.CRLF
  00001'020001-               LDA 0,CSTR1
  00002'040002-               STA 0,PNTR
  00003'030002-  LOOP:        LDA 2,PNTR
  00004'014002-               DSZ PNTR
  00005'024000-               LDA 1,CSTR
  00006'132433                SUBZ# 1,2,SNC
  00007'000002$               JMP DONE
  00010'151220                MOVZR 2,2
  00011'021000                LDA 0,0,2
  00012'024001$               LDA 1,C377
  00013'101002                MOV 0,0,SZC
  00014'101300                MOVS 0,0
  00015'123400                AND 1,0
  00016'177777                TYPET
  00017'000764                JMP LOOP
  00020'060111  INIT:         NIOS TTO
  00021'000757                JMP BGN
         000772'              .LOC .+750
  00772'040440  STRING:       .TXT * A
  00773'047526  VO
  00774'020116  N
  00775'042522  RE
  00776'052520  PU
  00777'020123  S
  01000'000000  *

         000020'              .END INIT
```

```
                              .TITL    AVON
                              .EXTD    CCRLF,.CRLF
                              .EXTN    BGN
                              .ENT     C377,DONE,TYPET,CRLF

                              .ZREL
00000-000377   C377:         377
00001-000007'  .TTTO:        TTTO
00002-177777   .BGN:         BGN
00003-006002$  DONE:         JSR 9.CRLF
00004-063077                 HALT
00005-002002-                JMP 0.BGN
       006001- TYPET=        JSR 9.TTTO

                              .NREL
00000'054406   CRLF:         STA 3,RCRLF
00001'020001$                LDA 0,CCRLF
00002'006001-                TYPET
00003'101300                 MOVS 0,0
00004'006001-                TYPET
00005'002401                 JMP 9RCRLF
       000001  RCRLF:        .BLK 1
00007'063611   TTTO:         SKPDN TTO
00010'000777                 JMP .-1
00011'061111                 DOAS 0,TTO
00012'001400                 JMP 0,3
                             .END
```

NUMBER DEFINITION

The number defining capability of the extended assembler has been expanded considerably over that of the basic assembler. The improvements help the user interface more easily with Data General's math library and floating point interpreter.

Decimal

To input decimal numbers using the basic assembler, it was first necessary to declare

.RDX   10

In addition to the .RDX pseudo-op the extended assembler allows the user to specify a decimal number at any point in his program by terminating a numeral string with a decimal point. However, no numeral may follow the decimal point without the number being interpreted as floating point. For example, in *any* radix, 10. will be interpreted as decimal 10 and be converted to octal 12 whereas 10.0 will be interpreted as a decimal floating point number. The decimal defining feature allows the programmer to combine decimal numbers in expressions with numbers of other radices. The following illustrates this.

Assembled Storage Word	Program Code
	.RDX 2
000152	101 + 101.
	.RDX 8
000246	101 + 101.
	.RDX 10
000312	101 + 101.

Floating Point

If a numeral string is followed by an 'E' or if the numeral string contains a decimal point followed by at least one more numeral or the letter 'E' the extended assembler will interpret the string as a floating point number. It will convert the string to a two word, floating point constant using the binary

fraction representation discussed in Appendix C of <u>How to Use the Nova</u>. This format is the one used by Data General's floating point interpreter (see write-up 093-000019). If numbers too large or too small to be represented are specified the assembler will regard them as errors and flag them with an N flag.

The following examples illustrate the definition of floating point constants.

Assembled Storage Words	Program Code
040420 000000	1.0
040426 041766	3.1415926
140420 000000	-1E0
040200 000000	+5.0E-1

The number following the 'E' is the decimal power of ten used to evaluate the number. The last example therefore implies

$$+(5.0)*(10)^{-1} = +0.5$$

Note: Although floating point constants may be used in data statements they are not permitted in expressions.

Double Precision

The math library provides for extensive manipulation of double precision numbers. These numbers are represented using two contiguous memory words (or two hardware accumulators) concatenated into a 32-bit string where the first word comprises bits 0 to 15 of the number and the second word bits 16 to 31 of the number.

```
0        15 16        31
|  word 1  |  word 2  |
```
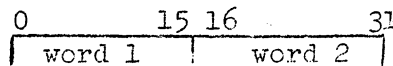
Bit 0 contains the sign and bits 1 to 31 contain the magnitude in two's complement notation. Double precision constants can be defined using the extended assembler by terminating a numeral string with a D. The numeral string, which may be signed, is then evaluated in the current radix, but as with single precision number definition no check is made for arithmetic overflow. The following strings convert as shown (assuming radix 8).

Assembled Storage Words	Program Code
000000	1D
000001	
177777	-1D
777777	
000001	200000D
000000	

Note: Double precision numbers cannot be combined in expressions.

It is also possible to specify double precision *decimal* numbers at any point in a program by using the decimal point followed by a 'D', but as with all double precision numbers they may not be combined in expressions. For example,

Assembled Words	Program Code
000004	262147.D
000003	
000001	100000.D
103240	

Bit Boundary Alignment

A facility for right justification of a single precision integer on a bit boundary is provided in the extended assembler. The specification of an integer in the current radix followed by

B *decimal number*

will cause the binary equivalent of the integer to be aligned at the bit boundary

designed by the decimal number. Thus the decimal number is limited to the
range 0 to 15. The statement takes the form

$$n \ \text{B} \ d$$

where n is a number in the current radix (usually octal) and d is a decimal
number specifying the bit boundary. The number is given the value

$$(n)_r * (2)_{10}^{(15-d)}$$

where r represents the current radix. The following strings are converted
as shown (radix 8).

Assembled Word (Binary)	Program Code
1 000 000 000 000 000	1B0
0 000 000 000 000 010	1B14
0 000 010 100 000 000	12B8

CONDITIONAL ASSEMBLY

The extended assembler provides a conditional assembly feature which allows
portions of a program to be assembled or to be by-passed by the assembler on the
basis of the evaluation of absolute expressions. Three pseudo-ops are used to
control the conditional assembly feature. They have the form

.IFE *Expression* (or .IFN *Expression*)

.

.

.ENDC

The expression in the .IFE (or .IFN) pseudo-op must be evaluable in pass 1 of
the assembly process. Otherwise it will be regarded as an error and flagged
with a K flag. That is, all symbols used in the expression must be absolute and
defined previous to the occurrence of the .IFE. If, when evaluated, the
expression *equals* zero, the statements following the .IFE will be assembled,
but when the evaluated expression does not equal zero all statements subsequent
to the .IFE pseudo-op up to the occurrence of an end conditional pseudo-op (.ENDC)

will be ignored. It is possible to specify the opposite situation by using the
.IFN pseudo-op with an expression. When using the .IFN pseudo-op subsequent
statements will be assembled only if the evaluated expression does *not* equal
zero, and will be by-passed when the expression equals zero.

Two further points should be kept in mind when using the conditional
assembly feature: 1) Conditionals may not be nested, *ie*, if a second .IF
pseudo-op is encountered before an .ENDC pseudo-op is found, the second .IF
will be ignored and will receive a K flag. 2) The pseudo-ops .END and .EOT
will not be ignored when imbedded in a section of conditionally assembled
code, *ie*, in the following example the .END will not be bypassed but will cause
the assembler to cease the assembly process.

```
.IFE    1
.END
.ENDC
```

APPENDIX A

Operating Procedure

The procedure used in assembling source code with the extended assembler is identical to that used with the basic assembler. However, two additional options for specifying the punched binary output are provided. These options cause the table of local symbols generated during assembly to be included in the punched output. This table of local symbols should be output only when the programmer intends to debug his program using the symbolic debugger since the binary tapes without local symbols are considerably shorter.

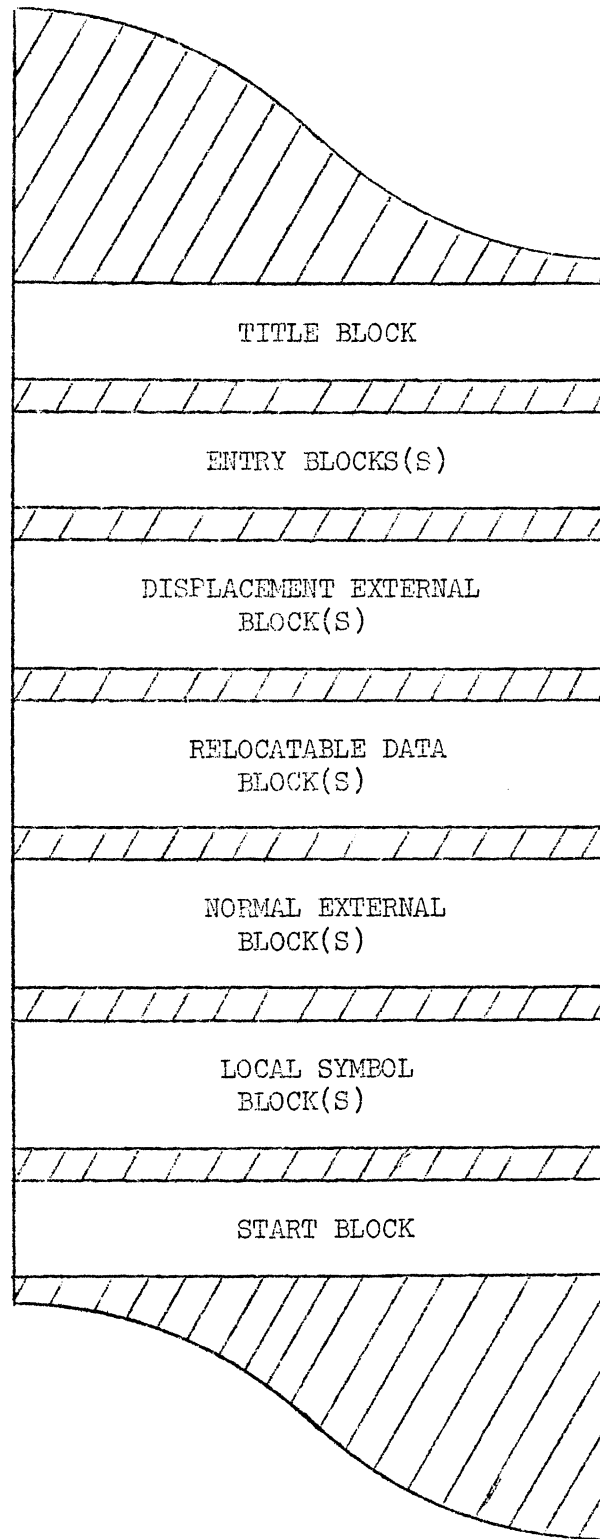Thus, when the assembler asks what form the binary output is to take, by typing,

BIN:

there are four possible responses whose effects are shown below.

RESPONSE	EFFECT
1	Output binary on the teletype *without* local symbols.
2	Output binary on the high speed punch *without* local symbols.
3	Output binary on the teletype *with* local symbols.
4	Output binary on the high speed punch *with* local symbols.

Like the basic assembler, the extended assembler punches its output in blocks separated by null characters. There are seven different types of blocks punched by the extended assembler which are distinguished by the code contained in the first word of each block. There is a specific order in which these various types are punched with all blocks of one type being punched together. For each program assembled the extended assembler will punch a Title Block, a Start Block, and at least one other block, but aside from the Title and Start Blocks no other type of block must necessarily appear in every program. The seven types of blocks in the order in which they would be punched if all were required are shown on the next page. The exact formats of each of these blocks can be found in Appendix C of the Relocatable Loader write-up (093-000039).

Order of Blocks Punched in Paper Tape

TITLE BLOCK

ENTRY BLOCKS(S)

DISPLACEMENT EXTERNAL
BLOCK(S)

RELOCATABLE DATA
BLOCK(S)

NORMAL EXTERNAL
BLOCK(S)

LOCAL SYMBOL
BLOCK(S)

START BLOCK

APPENDIX B

Error Mnemonics

FLAG	MEANING

A   Address error - Expression evaluates to something other than an absolute, normal relocatable, or page zero relocatable address. Page zero relocatable instruction references address outside page zero. Normally relocatable instruction references address outside the range of location counter relative addressing.

G   Error in declaration of an internal or external symbol.

K   Conditional assembly error - Expression used in .IFE or .IFN pseudo-ops is not evaluable in pass 1, or the .IFE or .IFN pseudo-op is nested within a previous conditional assembly statement.

N   Number specified is too large or too small to be represented as a floating point number.

R   Expression error - Expression does not evaluate to be absolute, relocatable, or byte pointer type relocatable, or expression mixes page zero and normal relocatable symbols incorrectly.

Z   Expression contains illegal symbol, (eg, an external, an op code, double precision number, or floating point number).