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

PROGRAM

The Macro Assembler

This document provides a brief description of the DGC Macro Assembler. This assembler is upward compatible with the RDOS Relocatable Assembler. It is written to take full advantage of the RDOS file system. Expansion of the symbol table and definition of macros is limited only by disk storage capacity.

CURRENTLY SCHEDULED
FOR OCT 15TH RELEASE

093-000081-00

The DGC Macro Assembler provides a number of major advancements over the Relocatable Assembler. In brief, these advancements include:

1. Expanded expression syntax that provides for implicit as well as explicit precedence. The class of operators has been expanded to include all logical comparison operators.

2. An assembly repeat feature for producing many lines of source from a simple repeat construct. This facility, given a repeat constant of zero, also provides for conditional assembly. Unlike the Relocatable Assembler, conditionals may be nested to any depth.

3. An assembler variable replacement scheme, allowing the programmer access to assembler variables using appropriate source line references. This facility enables the programmer to, for instance, define macros having no fixed argument list, generate unique labels, and access the assembler error flags.

4. A powerful macro facility which allows complete recursion as well as nested macro calls and which provides virtually no limitation upon the number of macro definitions. An extensive macro library is provided by DGC including macros for often used operations such as shifts, byte operations, logical operations, and signed comparisons.

5. Use of literal references with any memory reference instruction. All literals will be optimally resolved in page zero. Optionally, the assembler will eliminate all address errors by literal indirect references through page zero. Literals are not restricted to absolute numeric quantities and, in fact, may consist of any legitimate expression.

The DGC Macro Assembler is completely compatible with the DGC Relocatable Assembler. The additional capabilities of this assembler will be briefly summarized below.

Equivalence Redefinition

A most important addition of the Macro Assembler allows for equivalence redefinition. Given that the symbol appears to the left of an equal sign (=), it will be permissible to redefine its value. The evaluation will be similar to a FORTRAN assignment statement, *i.e.*, every occurrence of the symbol to the right of the equal sign implies the symbol's current value. The evaluation then *replaces* the current value with the new value. For

Equivalence Redefinition (Continued)

example, the following statements are legitimate and define "A" as given:

Statement	Value of A
A = 1	1
A = A+3	4
A = 2*A	10_8

Expressions

Six logical operators have been added to the set of available operators. Logical comparisons yield the numeric value 1 for true and 0 for false. In addition, implied precedence has been implemented. Note, however, that because of the left to right evaluation of the Relocatable Assembler the operators +, -, *, /, &, and ! have been given equal precedence. A complete list of operators is given below in order of decreasing precedence:

Precedence Level	Symbolic Operator	Operation
3 (highest)	B	Bit alignment
2	!	Logical OR
	&	Logical AND
	+	Addition
	-	Subtraction
	*	Multiplication
	/	Division
1 (lowest)	==	Equality
	<>	Inequality
	>	Greater than
	>=	Greater than or equal
	<	Less than
	<=	Less than or equal

To enable flexibility, parentheses can be used to explicitly determine the precedence of evaluation. Parenthesized expressions are, naturally, evaluated first. The following statements illustrate this evaluation.

```
000030 A=       10.+(7*2)
000330 B=       A+3B9
000001 C=       (B-A)B(16.-1)/(B-A)B15==1
000361 D=       C+(A+(B*C))
```

Do Loops

In addition to the .IF_.ENDC construct, conditional assembly as well as assembly source loops have been implemented using a new .DO_.ENDC construct. All source statements between the pseudo-ops .DO and .ENDC will be assembled the number of times specified by an absolute expression. The form of the .DO statement is

.DO *<absolute_expression>*

Do Loops (Continued)

All source lines between the .DO and the corresponding .ENDC are assembled <*absolute_expression*> times. Zero is a legal value, and provides for no assembly whatsoever.

A sixteen-word bit table can be generated by the following:

```
I=      0
        .DO     16.
        1BI
I=      I+1
        .ENDC
```

The assembly listing produced is:

```
      000000
      000020
00000 100000
      000001

00001 040000             1BI
      000002 I=          I+1
                         .ENDC
00002 020000             1BI
      000003 I=          I+1
                         .ENDC
00003 010000             1BI
      000004 I=          I+1
                         .ENDC
00004 004000             1BI
      000005 I=          I+1
                         .ENDC
00005 002000             1BI
      000006 I=          I+1
                         .ENDC
00006 001000             1BI
      000007 I=          I+1
                         .ENDC
00007 000400             1BI
      000010 I=          I+1
                         .ENDC
00010 000200             1BI
      000011 I=          I+1
                         .ENDC
00011 000100             1BI
      000012 I=          I+1
                         .ENDC
00012 000040             1BI
      000013 I=          I+1
                         .ENDC
00013 000020             1BI
      000014 I=          I+1
                         .ENDC
00014 000010             1BI
      000015 I=          I+1
                         .ENDC
00015 000004             1BI
      000016 I=          I+1
                         .ENDC
00016 000002             1BI
      000017 I=          I+1
                         .ENDC
00017 000001             1BI
      000020 I=          I+1
                         .ENDC
```

Do Loops (Continued)

Nested DO's are permitted. An $n**2$ bit table (from $n = 1$ to 4) can be generated by the following:

```
I=      1
        .DO     4
J=      0
WORD=   0
        .DO     I*I
WORD=   1BJ+WORD
J=      J+1
        .ENDC
I=      I+1

        WORD

        .ENDC
```

The table generated, in binary, is as shown below:

```
1
1111
111111111
1111111111111111
```

Assembler Variable Replacement

The Macro Assembler provides for string replacement of certain predefined symbols with assembler variable information. The occurrence of a replacement symbol in an assembler source statement is replaced by the appropriate string before being processed. Replacement symbols are always preceded by the character *$*. Thus the form is:

$<symbol>

The *<symbol>* is first matched against a table of predefined symbols. If recognized, the *$<symbol>* is replaced, before being processed, by an appropriate ASCII string. If the *<symbol>* is not recognized, it is searched for within the user symbol table itself. If found, the numeric value of the symbol, in ASCII, replaces the *$<symbol>* of the source statement. A table of predefined symbols is given below.

Symbol	Value
BREAK	Last assembler break character
FLAG	Last assembler error flag
LCMD	Location counter mode (0--> absolute, 1-> nrel, 2 ->zrel)
NOARG	Number of arguments of current macro
NOCAL	Number of calls of current macro
DEPTH	Current macro depth
PASS	Current assembler pass

Macro Definitions

Macro definitions take the form:

.MACRO <*macro_name*>

<*macro_definition_string*>!

Two special characters are interpreted within the macro definition string. The first is a single quote ('). This character is ignored and causes the next character to be stored as part of the definition without any interpretation. An example of the use of this character will be given below.

The second special character is an up arrow (↑). This character, if followed by a numeric (1-9) or an alphabetic (A-Z) is a representation of a formal argument. If the character following the "↑" is a numeric, *n*, ↑*n* will be replaced by the *n*th argument specified within the call. If the argument is an alphabetic, this single character symbol will be looked up at the time of the call and its value, *v*, will be used to replace the *v*th argument within the call.

Macro calls may appear anywhere within a source line and take the following form:

<*macro_name*> [<*actual_argument*$_1$>,<*actual_argument*$_2$>,...<*actual_argument*$_i$>...]

Substitution of actual arguments is accomplished by using *actual_argument*$_1$ to replace every occurrence of ↑1 (or ↑<*alpha*> where <*alpha*> has the value 1), <*actual_argument*$_2$> to replace every occurrence of ↑2, etc. If no formal arguments were specified within the definition, no arguments may be specified by the call. If more arguments are given by the call than specified, they are ignored.

A number of macro examples follow. Note the use of the added assembler features, the recursive property of the macro FACT and the use of the special character ' within VFD.

A very simple example, logical OR, is given below. The definition is:

```
; LOGICAL OR MACRO

; CALL:
;               OR       [<OP-0>,<OP-1>]

; WHERE THE RESULT IS:

;               <OP-0> .OR. <OP-1>


        .MACRO  OR

        COM     ↑1,↑1
        AND     ↑1,↑2               ; CLEAR "ON" BITS OF AC↑1
        ADC     ↑1,↑2               ; OR RESULT TO AC↑2
!
```

Macro Definitions (Continued)

Three separate calls assemble as:

```
                    OR [1,2]
00000 124000              COM      1,1
00001 133400              AND      1,2              ; CLEAR "ON" BITS OF AC1
00002 132000              ADC      1,2              ; OR RESULT TO AC2

                    OR [3,0]
00003 174000              COM      3,3
00004 163400              AND      3,0              ; CLEAR "ON" BITS OF AC3
00005 162000              ADC      3,0              ; OR RESULT TO AC0

                    OR [2,3]
00006 150000              COM      2,2
00007 157400              AND      2,3              ; CLEAR "ON" BITS OF AC2
00010 156000              ADC      2,3              ; OR RESULT TO AC3
```

A macro to compute $n!$ iteratively is given next. The form of the call is:

FACT [*<variable>*,*n*]

where the result is *<variable>* = $n!$

The definition is:

```
                .MACRO FACT

        ^2=     1
        I=1
                .DO      ^1
        ^2=     ^2*I
        I=I+1
                .ENDC
        !
```

A call for 5! expands as:

```
                FACT [5,J]
000001 J=       1
000001 I=1
000005          .DO      5
000001 J=       J*I
000002 I=I+1
                .ENDC
000002 J=       J*I
000003 I=I+1
                .ENDC
000006 J=       J*I
000004 I=I+1
                .ENDC
000030 J=       J*I
000005 I=I+1
                .ENDC
000170 J=       J*I
000006 I=I+1
                .ENDC
```

Macro Definitions (Continued)

The same computation can be made using the recursive prcperty of macros. Here the definition is:

```
                .MACRO   FACT

                .DO      ^1==1
        ^2=     1
                .ENDC

                .DO      ^1<>1
                FACT     [^1-1,^2]
        ^2=     (^1)*^2
                .ENDC

        %
```

The complete listing for a call of 6! is:

```
              FACT [6,J]
000000        .DO      6==1
          J=  1
              .ENDC

000001        .DO      6<>1
              FACT     [6-1,J]
000000        .DO      6-1==1
          J=  1
              .ENDC

000001        .DO      6-1<>1
              FACT     [6-1-1,J]
000000        .DO      6-1-1==1
          J=  1
              .ENDC

000001        .DO      6-1-1<>1
              FACT     [6-1-1-1,J]
000000        .DO      6-1-1-1==1
          J=  1
              .ENDC

000001        .DO      6-1-1-1<>1
              FACT     [6-1-1-1-1,J]
000000        .DO      6-1-1-1-1==1
          J=  1
              .ENDC

000001        .DO      6-1-1-1-1<>1
              FACT     [6-1-1-1-1-1,J]
000001        .DO      6-1-1-1-1-1==1
000001    J=  1
              .ENDC

000000        .DO      6-1-1-1-1-1<>1
              FACT     [6-1-1-1-1-1-1,J]
          J=  (6-1-1-1-1-1)*J
              .ENDC

000002    J=  (6-1-1-1-1)*J
              .ENDC

000006    J=  (6-1-1-1)*J
              .ENDC

000030    J=  (6-1-1)*J
              .ENDC

000170    J=  (6-1)*J
              .ENDC

001320    J=  (6)*J
              .ENDC
```

Macro Definitions (Continued)

The use of ↑<*letter*> is shown below. The macro ODD accepts a call having any number of arguments, producing a table of all the odd values. The definition is:

```
          ; THE FOLLOWING MACRO PRODUCES A TABLE OF ALL
          ; ODD INTEGERS WITHIN THE MACRO CALL LIST.

          ; CALL:
          ;                ODD       [<ARG-LIST>]


                    .MACRO   ODD

          I=        1
                    .DO      $NOARG
                    .DO      ^I-(^I/2*2)==1
                    ^I
                    .ENDC
          I=        I+1
                    .ENDC
          ;
```

A call generates the following:

```
                    ODD [1,2,3,4,5,6,7,8.,9.]
      000001 I= 1
      000011    .DO   $NOARG000011
      000001    .DO   1-(1/2*2)==1
00000 000001    1
                .ENDC
      000002 I= I+1
                .ENDC
      000000    .DO   2-(2/2*2)==1
                2
                .ENDC
      000003 I= I+1
                .ENDC
      000001    .DO   3-(3/2*2)==1
00001 000003    3
                .ENDC
      000004 I= I+1
                .ENDC
      000000    .DO   4-(4/2*2)==1
                4
                .ENDC
      000005 I= I+1
                .ENDC
      000001    .DO   5-(5/2*2)==1
00002 000005    5
                .ENDC
      000006 I= I+1
                .ENDC
      000000    .DO   6-(6/2*2)==1
                6
                .ENDC
      000007 I= I+1
                .ENDC
      000001    .DO   7-(7/2*2)==1
00003 000007    7
                .ENDC
      000010 I= I+1
                .ENDC
      000000    .DO   8.-(8./2*2)==1
                8.
                .ENDC
      000011 I= I+1
                .ENDC
      000001    .DO   9.-(9./2*2)==1
00004 000011    9.
                .ENDC
      000012 I= I+1
                .ENDC
```

Macro Definitions (Continued)

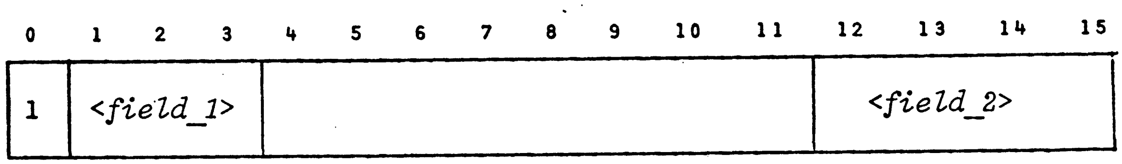
A powerful macro, used to associate a specified field layout with a given name, is now described. The macro VFD is used to define a new macro named as the first argument in the call to VFD. Subsequent use of the name given in the VFD call generates a 16-bit storage word having a primary value to which fields are assembled as described in the call to VFD. The call is of the form:

VFD [<*type_name*>,<*primary_value*>,<*field*$_1$ *right bit*>,<*field*$_1$ *mask*>,...
<*field*$_i$ *right bit*>,<*field*$_i$ *mask*> ...]

The 3rd, 5th, ... arguments specify the rightmost bit positions of the 1st, 2nd, ... fields. The 4th, 6th, ... arguments specify the field masks for the 1st, 2nd, ... fields. To assemble the fields in the proper bit positions, with overflow and field zero checking, a call is made of the form:

<*type_name*> [<*field_1* >,<*field_2*> ...]

The example below defines a <*type_name*> of XYZ. This name is for words of the following layout:



The definition of VFD is:

<u>Macro Definitions</u> (continued)

```
        .MACRO VFD

        I=4
                        .MACRO ^1

                        VALU=^2
                        J=1
'!
        .DO $NOARG/2-1
                        .MACRO ^1 *

                        .IFN ^I>='^J
                        MASK=^I
                        DATA='^J
'!
        I=I-1
                        .MACRO ^1*

                             .DO 15.-^I
                             MASK=MASK*2
                             DATA=DATA*2
                             .ENDC
'!
        I=I+1
                        .MACRO ^1 *

                             .IFN VALU&MASK
                             ERROR [FIELD NON-ZERO]
                             .ENDC

                             .IFE VALU&MASK
                             VALU=(VALU&(-MASK-1))+DATA
                             .ENDC
                        .ENDC

                        .IFE ^I>='^J
                        ERROR[FIELD OVERFLOW]
                        .ENDC
'!
        I=I+2
                        .MACRO ^1 *

                        J=J+1
'!
        .ENDC
                        .MACRO ^1*

                        VALU
'!
!
```

* Definition of a macro previously defined causes the latest <*definition_string*> to be appended to the previous string.

The call to define XYZ produces:

```
VFD [XYZ,100000,3,7,15.,17]
  I=4

        .MACRO XYZ

        VALU=100000
!       J=1
  .DO $NOARG00006/2-1
        .MACRO XYZ

        .IFN 7>=^J
        MASK=7
!       DATA=^J
  I=I-1

        .MACRO XYZ

                .DO 15.-3
                MASK=MASK*2
                DATA=DATA*2
!               .ENDC
  I=I+1
        .MACRO XYZ

                .IFN VALU&MASK
                ERROR [FIELD NON-ZERO]
                .ENDC

                .IFE VALU&MASK
                VALU=(VALU&(-MASK-1))+DATA
                .ENDC
        .ENDC

        .IFE 7>=^J
        ERROR[FIELD OVERFLOW]
!       .ENDC
  I=I+2

        .MACRO XYZ

!       J=J+1
  .ENDC
        .MACRO XYZ

        .IFN 17>=^J
        MASK=17
        DATA=^J
!
  I=I-1
        .MACRO XYZ

                .DO 15.-15.
                MASK=MASK*2
                DATA=DATA*2
                .ENDC
!
  I=I+1
        .MACRO XYZ

                .IFN VALU&MASK
                ERROR [FIELD NON-ZERO]


                .ENDC

                .IFE VALU&MASK
                VALU=(VALU&(-MASK-1))+DATA
                .ENDC
        .ENDC

        .IFE 17>=^J
        ERROR[FIELD OVERFLOW]
        .ENDC
!
  I=I+2
        .MACRO XYZ

        J=J+1
!
  .ENDC
        .MACRO XYZ

        VALU
!
```

And, finally, a call of XYZ for fields having values of 2 and 10 gives:

```
        XYZ [2,10]
100000          VALU=100000
000001          J=1
000001          .IFN 7>=2
000007          MASK=7
000002          DATA=2
000014                  .DO 15.-3
000016                  MASK=MASK*2
000004                  DATA=DATA*2
                        .ENDC
000034                  MASK=MASK*2
000010                  DATA=DATA*2
                        .ENDC
000070                  MASK=MASK*2
000020                  DATA=DATA*2
                        .ENDC
000160                  MASK=MASK*2
000040                  DATA=DATA*2
                        .ENDC
000340                  MASK=MASK*2
000100                  DATA=DATA*2
                        .ENDC
000700                  MASK=MASK*2
000200                  DATA=DATA*2
                        .ENDC
001600                  MASK=MASK*2
000400                  DATA=DATA*2
                        .ENDC
003400                  MASK=MASK*2
001000                  DATA=DATA*2
                        .ENDC
007000                  MASK=MASK*2
002000                  DATA=DATA*2
                        .ENDC
016000                  MASK=MASK*2
004000                  DATA=DATA*2
                        .ENDC
034000                  MASK=MASK*2
010000                  DATA=DATA*2
                        .ENDC
070000                  MASK=MASK*2
020000                  DATA=DATA*2
                        .ENDC
000000                  .IFN VALU&MASK
                        ERROR [FIELD NON-ZERO]
                        .ENDC
000001                  .IFE VALU&MASK
120000                  VALU=(VALU&(-MASK-1))+DATA
                        .ENDC
                .ENDC
000000          .IFE 7>=2
                ERROR[FIELD OVERFLOW]
                .ENDC
000002          J=J+1
000001          .IFN 17>=10
000017          MASK=17
000010          DATA=10
000000                  .DO 15.-15.
                        MASK=MASK*2
                        DATA=DATA*2
                        .ENDC
000000                  .IFN VALU&MASK
                        ERROR [FIELD NON-ZERO]
                        .ENDC
000001                  .IFE VALU&MASK
120010                  VALU=(VALU&(-MASK-1))+DATA
                        .ENDC
                .ENDC
000000          .IFE 17>=10
                ERROR[FIELD OVERFLOW]
                .ENDC
000003          J=J+1
00000 120010    VALU
```

Literals

Literals are permitted on all memory reference instructions. The format is:

<mem_reference>[*<ac>,*] = *<expression>*

When assembled, the instruction will reference a memory location whose contents will be the expression given. To accomplish this, all literals will be "dumped" in page zero, using the first .ZREL location available after pass 1.

An extension of the literal concept is optionally available to resolve program address errors. All address errors detected on Pass 2 will be resolved using page zero address constants provided by the assembler.

Listing Options

The Macro Assembler provides, optionally, a number of listing options. These options include:

1. The octal output from .TXT can be suppressed after two bytes (one word).
2. Conditional statements that are not assembled can be suppressed from the listing.
3. Macro expansions can be suppressed from the listing.
4. An expanded address field can be printed, giving the address of all memory reference instructions relative to the current location counter.