Identification

Description of macro calls.

Jean C. Scholtz

Introduction

Pass one of the EPL compiler produces a series of macro calls.
These calls consist of operators, scales (an extravagant gener-
alization of PL/I idea of scale) and various fields. The first
field is the name of the variable or constant specified by the
user in the EPL program. The second field is the created name
for this variable or constant. The remaining fields give in-
formation about this variable or constant. A macro call has
the following form:

    operator/scale   field 1,.......,field 10

The number of fields appearing in the call depends upon the
operator; however, the usual number of fields is ten. A
more detailed discussion of this will appear in the explana-
tion of the fields. A typical line of macro code would be:

    dffx   ,xx0058,17,0,xxx,int,auto,0,1,0

In this example "df" is the operator, "fx" is the scale and
there are ten fields present, the first field being null.

Classification of Macro Calls

The macro calls are divided into classes according to their
operators. The four classes and a brief description of each
follow.

   A.   Binary Operators - These consist of arithmetics
        operators, relational operators, boolean operators,
        and maximum, minimum and mod operators. Simple IF*
        instructions are included in this category. These
        are such relations as IF Y < X, IF Y = X, etc.

---

* Capital letters will be used to indicate EPL commands. An
  underlined word will signify macro code in cases where it
  might not be obvious from the context.

B. Unary Storage Operators - This includes load and
    store instructions, CALL instructions, GO TO
    instructions, ON conditions, VALIDATE instructions,
    difficult IF constructions and the LENGTH function.

C. Unary Accumulator Operators - This includes such
    operators as the unary plus, the not operator,
    the change sign instruction and all conversions.

D. Operators Needing External Names - In this cate-
    gory are the operators used for defining all
    constants, labels and variables. The entry macro
    which will be discussed later also comes under
    this category.

Operators

Most of the abbreviations for the operators are fairly mnemonic.
Following is a table of the operators (by classification) along
with their meanings.

A. Binary Operators

    ad   - add
    sb   - subtract
    ml   - multiply
    dv   - divide
    nd   - and
    or   - or
    eq   - equal
    ne   - not equal
    ct   - catenation
    gt   - greater than
    lt   - less than
    ge   - greater than or equal to
    le   - less than or equal to
    ifge   - if greater than or equal to
    ifle   - if less than or equal to
    ifeq   - if equal to
    ifne   - if not equal to

iflt   - if less than

ifgt   - if greater than

md   - mod function

mx   - maximum

mn   - minimum

B. Unary Storage Operators

la   - load address accumulator

sa   - store address accumulator

ld   - load

st   - store

lg   - length

on   - on conditions

sg   - signal

rv   - revert

validate - validate option

if   - difficult if statement (not just simple relational tests)

go   - go to

gogo   - go to (used for a transfer to a label variable whose value is in the current block)

call   - call

ls   - load the size (in words)

gf   - generate the effective address

C. Unary Accumulator Operators

ch   - change sign

nt   - not

ab   - absolute value

sn   - sign function

af   - unary plus

D. Operators Needing External Names

df   - define variables

dc   - define constants

entry   - entry

Scales

The scale abbreviations are also mnemonic. A full list of the possible scales follows.

pt	-	pointer
bs	-	bit string
cs	-	character string
fl	-	floating
fx	-	fixed
lb	-	label
sx	-	structure
cd	-	condition name
fi	-	file
aa	-	temporary location for the address accumulator
xx	-	undefined
db	-	dimension bounds

Discussion of Fields

The macro code usually consists of zero to ten fields. More than ten fields may occur in the case of subscripting. Table 1 is a diagram of the fields in the order in which they would appear in pass one output.

The first field appears only when the operator is df or dc. When any other operator appears the first field is missing. The original value is the name of this variable in the EPL program. When the variable or constant is generated by the compiler the first field is null.

The created name will be of the form xxNNNN where N is some digit. This will be the name the compiler has chosen for this variable. All further references to this variable will be by the created name.

The actual length in bits appears if the length is declared to be constant. * would appear if the length were so declared. Otherwise the length field contains the name of a subroutine which will compute the length in bits or characters.

The offset field is so known for historical reasons. This field is usually zero. If the variable is an adjustable length string then the adjustable flag is set and this field contains adj. If the variable is external static with the initial attribute then this field contains esi.

TABLE 1

MACRO FIELDS

$$\left\{ \begin{array}{l} \text{original value} \\ \text{of constant/} \\ \text{original name} \\ \text{of variable} \end{array} \right\} \quad \left\{ \begin{array}{l} \text{created} \\ \text{name} \end{array} \right\} \quad \left\{ \begin{array}{l} \text{length} \\ \text{in bits} \end{array} \right\} \quad \left\{ \begin{array}{l} \text{offset:} \\ \underline{0} \text{ or } \underline{adj} \\ \text{or } \underline{esi} \end{array} \right\} \quad \left\{ \begin{array}{l} \underline{xxx} \\ \text{or } \underline{var} \end{array} \right\} \quad \{\text{scope}\}$$

$$\left\{ \begin{array}{l} \text{storage} \\ \text{class} \end{array} \right\} \quad \left\{ \begin{array}{l} \text{\# of} \\ \text{dimensions} \end{array} \right\} \quad \left\{ \begin{array}{l} \text{block} \\ \text{level} \end{array} \right\} \quad \left\{ \begin{array}{l} \text{\# of} \\ \text{substructures} \end{array} \right\}$$

The fifth field contains <u>var</u> when the variable being con-
sidered in a varying string. At all other times it contains
<u>xxx</u>.

The scope field has five possibilities: <u>int</u>, <u>ext</u>, <u>con</u>, <u>mos</u>
or <u>par</u>. <u>int</u> indicates that the scope of this variable is
internal. Similarly, <u>ext</u> denotes the fact that it is external.
If <u>con</u> appears the variable is a constant. Seeing <u>mos</u> in this
field, one knows that the variable is a member of a structure.
If <u>par</u> appears the variable is a parameter. When <u>par</u> appears
an integer will also appear. This indicates which parameter is
being considered. For example the EPL statements:

        test:proc (a,b,c);
            dcl (a,b,c) fixed binary;

would produce the following macro code:

        dffl test,xx0024,27,0,xxx,ext,entr,0,0,0
        begin
        entry test,xx0024,27,0,xxx,ext,entr,0,0,0

        dffx a,xx0026,17,0,xxx,par1,xxxx,0,1,0
        dffx b,xx0027,17,0,xxx,par2,xxxx,0,1,0
        dffx c,xx0028,17,0,xxx,par3,xxxx,0,1,0

The storage class is one of the following: <u>auto</u> (automatic),
<u>cont</u> (based), <u>stat</u> (static), <u>entry</u> (entry) or <u>xxxx</u> (none).
Though entry is not an EPL storage class it appears here.
Should entry data be introduced into EPL, this would cause
trouble.

The remaining three fields are fairly self-explanatory. The
eighth field indicates the number of dimensions. The next field
specifies the block level where the variable we are presently
looking at was declared. Level 0 names are built in or external
entries to the program being compiled. Constants appear at
level 1. The last field indicates how many substructures this
structure has.

Consider the following examples:

    (1) dfpt   p,xx0026,72,0,xxx,int,auto,0,1,0

            p is defined to be a pointer. The compiler will use
            xx0026 to refer to p. p is 72 bits in length, the
            scope is internal, p uses automatic storage and is
            declared at the first block level.

(2) gfsx   xx0025,0,0,xxx,int,cont,0,1,3

This macro is to generate the effective address of a
structure which has the created name xx0025. This is
a based structure declared at block level one and
having three substructures.

When the operator appearing in the macro is a binary operator,
the number of fields will increase from the usual ten to twelve.
The first two fields contain the length and offset of the
variable in the accumulator. The remaining ten fields are the
usual fields that appear.

Special Macros

Certain code output by pass one deserves more attention since
it differs slightly from the usual forms of code.

The macro use appears with one of the following replacing the
usual fields: autoinit, statinit, statinitint, statinitext,
contbds or main. These instructions indicate to pass two of
the compiler to use routines for setting initial values into
automatic storage and static storage, and for calculating
adjustable bounds of lengths. There is a distinction between
static storage that is internal in scope and that which is
external. When the static storage is internal int will not
always appear following statinit. It is, however, essential
that ext follow statinit when the variable is external in
scope. A second field will then appear - the created name
of the variable. The instruction use main indicates to pass
two to return to the main sequence of code. The macro that"s
all is used in conjunction with the use macro and it merely ends
the bounds calculating routine. Examples:

(1) use statinitext,   xx0027

(2) use main

The bend macro indicates the end of a procedure block or a
begin block. It has no fields. Likewise, begin usually has
no fields and indicates the start of a procedure or begin
block. Begin for an on-unit has one field, "on". An entry
to any procedure is denoted by the entry macro. This code
will be compiled from the procedure statement at the beginning
of the EPL program or from a DECLARE statement within the
program. In particular PROCEDURE maps into begin then entry.

The macro if has as its argument a label. This contains the
location which is transferred to if the bit string contained
in the accumulator is not zero.

The macros <u>enable</u> and <u>disable</u> appear when condition prefixes
are used in the EPL procedure. The procedure

```
a:proc;
(nozerodivide):b:begin;
                const=6/0;
                end b;
end a;
```

would produce (among much other code) the following macros:

```
...
disable   zerodivide
...
begin
...
bend
enable. zerodivide

bend
```

The Address Accumulator

Generally speaking, pass one of the EPL compiler generates one
address code, which assumes an "accumulator" capable of holding
a datum of any scale. In accessing elements of aggregates a
second "address accumulator" is involved. The address accumu-
lator is set by gf (generate effective address) or <u>la</u> (load
address accumulator). To perform this assignment:

```
DCL     la,2b,3c;a.b.c = y;
```

this series of (schematic) macros would appear:

```
...
ldfl     y,......
gfsx     a,....,int,auto,0,1,1
gfsx     b,....,mos,xxxx,0,1,1
stfl     c,....,mos,xxxx,0,2,0
```

The regular accumulator remains unchanged while the address
accumulator is used to walk through the structures. That is,
first y is loaded into the accumulator. Then since c is a
member of a structure it must be located by finding the address
of the outer members of the structure. The address accumulator
is used to locate a and then b, giving access to c into which
is stored the contents of y.

The address of based data is understood to be in the address accumulator. Thus $x = p \rightarrow a$; yields

```
lapt      p,......
ldfl      a,....,int,cont,....
stfl      x,......
```

Subscripting is done by first getting the address of the element into the address accumulator. The element itself can then be accessed as if it were controlled; in fact, a nameless controlled variable sharing the attributes of the element is invented for this purpose. Subscripting is indicated by following a gf by a special subs macro. It is done out of line. For example these statements:

DCL   a(10,10);   a(i,j) = b;

would generate the following code:

```
gffl      a,....,auto,2,1,0
subs      2,i,j
saaa      z,....,(temporary storage for address accumulator)
ldfl      b,......
laaa      z,......
stfl      ......,cont,0,1,0
```

The subs macro appears after the gf macro only when the number of dimensions is greater than zero. The first field following subs is the number of subscripts. The remaining fields are locations which contain the value of the subscripts. Here one should note that the number of fields may exceed the usual ten. Subscripts will always be default dinary integers at the current block level.

Procedure Options

There are several macros which are the procedure options mastermode, executeonly, callback, rename, system and validate. (See MSPM BP.0.02) This code appears at the beginning of pass one output. Mastermode, executeonly, callback, and system have no arguments. Validate gives the name of the validation procedure with scale and all fields. Rename has two fields, the original and new names.

Compilation of Some EPL Statements

Some EPL statements warrant special attention with regard to
the way they are compiled by pass one of the EPL compiler.

A DO statement of sufficient difficulty generates among other
code a special pair of marcos. The first is dofl or dofx. The
argument of this marco is the name of the place whose value
determines the sense of the test for termination of the do loop.
The second macro is outlb. Its argument is the label of the
place to transfer to upon exit from the loop. For example the
statement: Do i = 1 to 100; would produce (among much other
code) the following macros:

      dofx   xx0037,17,0,xxx,int,auto,0,1,0
      outlb   xx0032,144,0,xxx,con,xxxx,0,1,0

The created name xx0037 is used to hold the increment for the
do loop.

When a conversion is specified by the procedure the pass one
output is this:

      scale/scale   field 1,....,field 4

The first scale is the current form of the variable. The second
scale is the form to which the variable is to be converted. The
four fields are the lengths and offsets of the variable before
and after conversion. For instance the statements DCL A; A = 1;
produce the following macro code:

      dffl   a,xx0025,27,0,xxx,int,auto,0,1,0
      dcfx   1,xx0030,4,0,xxx,con,xxxx,0,1,0
      ldfx   xx0030,4,0,xxx,con,xxxx,0,1,0
      fxfl   4,0,27,0
      stfl   xx0025,27,0,xxx,int,auto,0,1,0

Here the fixed number 1 of length 4 bits is converted to a
27 bits floating point number and the result stored in A.

In the compilation of a simple IF statement a pair of macros
is generated. The first macro is if. This specifies the
relation to be tested. The second macro is golb. This indi-
cates the place to which control passes if the relationship
tested by the if is not true. For example, DCL(Z,Y) FIXED BIN;
IF Z = Y THEN GO TO LEND; compiles into:

```
dffx        z,xx0026,17,0,xxx,int,auto,0,1,0
dffx        y,xx0027,17,0,xxx,int,auto,0,1,0
ldfx        xx0026,17,0,xxx,int,auto,0,1,0
ifeqfx      17,0,xx0027,17,0,xxx,int,auto,0,1,0
golb        xx0031,144,0,xxx,con,xxxx,0,1,0
golb        xx0033,144,0,xxx,con,xxxx,0,1,0
dclb        ,xx0031,144,0,xxx,con,xxxx,0,1,0
...
dclb        lend,xx0033,144,0,xxx,con,xxxx,0,1,0
```

In this example it is seen that control is passed to the next
statement if the IF relation is false.

As a further help in following through the pass one code it
should be noted that the output occurs in blocks - each block
corresponding to a statement from the original EPL procedure.
That is, each semicolon in the EPL procedure is mapped into a
blank line in the macro code.