Identification

Declarations and Data Organization in EPL
D. B. Wagner

Purpose

The present Section describes in awesome detail the actions
of Pass 1 and Pass 2 of EPL in dealing with management
and definition of variables, aggregates, temporaries,
etc. The average user will have no interest in this Section;
it is intended for the use of EPL maintainers only, and
even for them only as a reference. See BN.6.00 for the
terminology used.

Non-string Scalars

The following data-types do not require specifiers. Each
simply requires a block of storage 1, 2, or 6 words long.
See BP.2.01 for details of implementation of these data-types;
the present discussion concerns only how their storage
is managed.

        floating variables
        fixed variables
        pointer variables
        label variables

If Pass 1 encounters a declaration for one of these data-types,
or if it needs a temporary variable with one of these
data-types, it generates a macro of the following form:

    dfxx name,alias,bits,offset,xxx,scope,class,0,level,0

Here xx gives the data-type:

        fl   floating
        fx   fixed
        pt   pointer
        lb   label

Most of the macro fields are as described in BN.2.02.
Name is the name used for the variable in the source program
(or null for a temporary), alias is a unique name generated
by Pass 1 for the variable, bits is the precision of a
fixed or floating variable, a constant 72 for a pointer
variable, or a constant 144 for a label variable. [This

constant should be 216 for a label variable, since labels
are now 6 words long instead of 4. However Pass 2 ignores
this field for labels so it doesn't really matter.] Offset
is either 0 or "esi". "Esi" means "external static initial";
this issue is discussed elsewhere. Level is the block
level at which the declaration occurred. The fields scope
and class differ according to the storage class of the
variable: they are mentioned in the following discussions
of the various storage classes.

1.   For a non-string scalar parameter, Pass 1 generates the macro

          dfxx name,alias,bits,0,xxx,par n,xxxx,0,1,0

      where n is the parameter number. Pass 2 totally ignores
      this macro.

2.   For an automatic non-string scalar, Pass 1 generates the
      macro,

             dfxx name,alias,bits,0,xxx,int,auto,0,level,0

Pass 2 assigns sufficient storage for the variable in
the current block. This storage is at an even location
if the variable is other than single-word arithmetic.
Pass 2 generates the following eplbsa line to associate
alias with its assignment of storage.

          equ alias,loc   name

      Loc is the stack location assigned to the variable. Name
      is given as a comment for convenience.

3.   For an internal static non-string scalar, pass 1 generates
      the macro:

             dfxx name,alias,bits,0,xxx,int,stat,0,level,0

See BN.5.00. for the implementation of internal static
storage. Pass 2 assigns storage to the variable in the
procedure's block of iternal static storage at <stat_>|<segame>.
This storage is at an even location if the variable is
other than single-word arithmetic. Assignment of storage
in the procedure's internal static storage block begins
with location 0. Pass 2 generates the following eplbsa
line to associate alias with its assignment of storage.

```
equ alias,loc name
```

Loc is the location assigned to the variable. Name is
given as a comment for convenience.

4.   For an external static non-string scalar without the
      initial attribute, Pass 1 generates the macro

```
dfxx name,alias,bits,0,xxx,ext,stat,0,level,0
```

See BP.4.00 for the implementation of external static
storage. Pass 2 switches to a code stream (see BN.6.01)
outside of the any executable code stream and generates
the following eplbsa code:

```
.yn:      dec          nwords
          dec          0
          segref       stat_,name (datmk_(.yn))
          link         alias,name
```

Here .yn is a unique symbol created by Pass 2, and nwords
is the number of words which the variable occupies, adjusted
up to an even number. [There seems to be no reason for
this adjustment.] The procedure datmk_ is used to "grow"
storage: it is described in great detail in BP.4.01.

5.   For a controlled, based non-string scalar without the
      initial attribute, Pass 1 generates the macro

```
dfxx name,alias,bits,0,xxx,int,cont,0,level,0
```

which Pass 2 ignores.

Non-adjustable Non-varying Strings

Strings require specifiers and dope and consequently are
more difficult to compile than other scalars. See BP.2.02
for details of specifiers: they contain its pairs which
can only be created at execution time.

If Pass 1 encounters a declaration for a non-adjustable
non-varying string, or if it needs a non-varying string
temporary for its own use, it generates a macro of the
following form.

```
dfxx name,alias,bits,offset,xxx,scope,class,0,level,0
```

Here xx gives the data-type:

    bs   bit-string
    cs   character-string

Most of the macro fields are described in BN.2.02. Name is the name used for the variable in the source program (or null for a temporary). Bits is the length of the string in bits. Offset is either "0" or "esi". "Esi" means "external static initial"; this issue is discussed elsewhere. Alias is a unique name generated by Pass 1 for the variable. Level is the block level at which the declaration was encountered. The fields scope and class indicate the storage class and are mentioned in the discussions of storage classes which follow.

1.   For a non-adjustable non-varying string parameter, Pass 1 generates the following macro:

        dfxx name,alias,bits,0,xxx,par n,xxxx,0,1,0

Here n is the parameter number. Pass 2 need only ignore this macro. However it does generate the label and transfer for jumping into and back out of the prologue code sequence (see BN.6.01). Thus one sees in the code wasted instructions such as,

        p1.4:   tra p1.5

2.   For an automatic non-adjustable non-varying string, Pass 1 generates the macro

        dfxx name,alias,bits,0,xxx,int,auto,0,level,0

Pass 2 allocates storage in the current block's stack frame for the specifier and data for the string. It compiles the dope into the procedure. It compiles into the prologue code sequence the necessary instructions to create the specifier at block entry.

The layout of the string in the stack will be as follows:
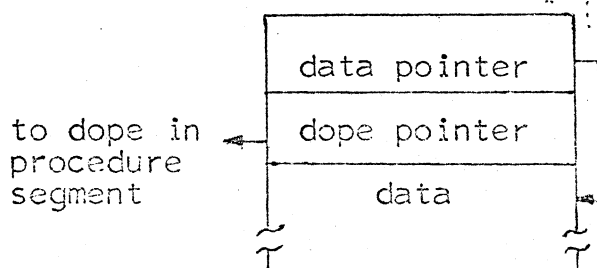
Pass 2 generates, outside of any executable code sequence:

```
        equ   alias,loc        name
.iap:dec  0
        vfd   9/160,27/bits
```

And in the prologue code sequence:

```
eapbp        sp|alias+4,n
stpbp        sp|alias+0,n
eapbp        .iap,n
stpbp        sp|alias+2,n
```

Here loc is the location assigned to the variable in the
stack. [The modifier ",n" on the instructions above means
"no modifier." It is normally left off in eplbsa code.
This is probably a harmless leftover from early misunder-
standings. The symbol .iap is a unique symbol created
by Pass 2.

3.   For a controlled, based non-adjustable non-varying string,
      Pass 1 generates the following macro:

```
dfxx name,alias,bits,0,xxx,int,cont,0,level,0
```

Pass 2 allocates space in the current block's stack frame
for the string specifier. It compiles the dope into the
procedure segment outside of any executable code sequence.
It compiles into the prologue code sequence the code to
set up half the specifier, the dope pointer, at entrance
to the current block. The data pointer in the specifier
is set to point to a generation of the variable when it
is accessed (see BN.6.03.). [As will be mentioned elsewhere,
this particular implementation of based items with specifiers
causes some unfortunate restrictions in the ways they
can be used in calls]

Pass 2 compiles, outside of any executable code sequence,
the following eplbsa code:

```
        equ   alias,loc        name
.iap:dec  0
        vfd   9/160,27/bits
```

And in the prologue code sequence:

```
eapbp        .iap
stpbp        sp|alias+2,n
```

Here loc is the location assigned to the specifier in the
current block's stack frame. The symbol .iap is a unique
symbol created by Pass 2.

4.   For an <u>internal</u> <u>static</u> non-adjustable non-varying string,
      Pass 1 generates the following macro:

        df<u>xx</u> <u>name</u>,<u>alias</u>,<u>bits</u>,0,xxx,int,stat,0,<u>level</u>,0

See BN.5.00. for the implementation of internal static
storage. Pass 2 assigns sufficient space for the string
and its specifier in the procedure's internal static storage.
It generates code to create the specifier in the "internal
static specifiers" code sequence. The layout of the string
in internal static storage is the same as that of an automatic
string in the stack, as diagrammed above.

      Pass 2 generates, outside of any executable code sequence,

```
              equ       alias,loc          name
  .dvn:       dec       0
              vfd       9/160,17/bits
```

      And in the "internal static specifiers" code sequence,

```
        eapap     lp|.is,*
        eapbp     ap|0
        adbbp     alias+4,du
        stpbp     ap|alias
        eapbp     .dvn
        stpbp     ap|alias+2
```

The symbol .<u>is</u> is the linkage address of the procedure's
internal static storage"; see BN.6.01. <u>Loc</u> is the location
assigned to the variable in this internal static storage.
The symbol .dv<u>p</u> is a unique symbol created by Pass 2.

5.   For an <u>external</u> non-adjustable non-varying string without
      the initial attribute Pass 1 generates the macro:

        df<u>xx</u> <u>name</u>,<u>alias</u>,<u>bits</u>,0,xxx,ext,stat,0,<u>level</u>,0

See BP.4.00 for the implementation of external static
storage. Pass 2 compiles, outside of any executable code
sequence, the dope and the code for setting up the external
variable on first reference. It allocates storage in
the procedure's internal static storage for the string
specifier, and compiles, in the "internal static specifiers"
code sequence, the code to initialize the specifier.

Pass 2 compiles, outside of any executable code sequence,

```
            equ        alias,loc              name
.dvn:       dec        0
            vfd        9/160,27/bits
.ym:        dec        nwords
            dec        0
            segref     stat_,name(datmk_(.ym))
            arg        name
```

And in the "internal static specifiers" code sequence,

```
        eapap      lp|.is,*
        eapbp      name,n
        stpbp      ap|alias+0
        eapbp      .dvn
        stpbp      ap|alias+2
```

Nwords is the number of words required by the string,
adjusted up to an even number of words. [It is not clear
why this adjustment is made.] Loc is the location assigned
to the string's specifier in the procedure's internal
static storage. The symbol .dvn and .ym are unique symbols
created by Pass 2. [The "arg name" above is another superfluous
remnant of the good old days of BSA.]

Non-adjustable Varying Strings

See BN.5.00 for the implementation of varying strings
in EPL. A varying string's specifier contains a third
its pair pointing to a free storage area where data is
kept for the string.

An anomaly concerning varying strings is that they must
be initialized to zero length before being assigned values.
Furthermore all the automatic varying strings in a block
must be cleared when the block is terminated. The jobs
of initializing and clearing varying strings and aggregates
containing them are performed through calls to the library
procedures varst_$zero and varst_$clear, described in
BN.7.02. Two internal subroutines compiled into each
program which needs them, .v1 and .v2, serve as interfaces
to these library procedures. They are called where they
are needed (normally using an eax 1 followed by a tsx0).
The detailed discussions given below for the various storage
classes show precisely how they are called in various
cases. The code for .v1 and .v2 is as follows:

```
.v1:        stpsp           sp|.u0+2
            asx7            sp|.u0+3
            ldaq            =v18/2,54/0
            staq            sp|.u0
            call            <varst_>|[zero](sp|.u0)
            tra             0,0
.v2:        stpsp           sp|.u0+2
            asx7            sp|.u0+3
            ldaq            =v18/2,54/0
            staq            sp|.u0
            call            <varst_>|[clear](sp|.u0)
            tra             0,0
```

The symbol .u0 is the stack location of a block of "utility" storage used in many places in the compiled code. It is available by the same name at all block levels. See BN.6.01.

   [The instructions above,

```
        stpsp           sp|.u0+2
        asx7            sp|.u0+3
```

are unacceptable because the second will cause an overflow fault when the stack grows longer than $2**17$ words. These instructions should be replaced by,

```
        eapbp           sp|0,7
        stpbp           sp|.u0+2
```

I am indebted to C. G. Garman for this and several other problems with overflow faults.]

1.   For an automatic non-adjustable varying string, Pass 1 generates the following macro:

        dfxx name,alias,bits,0,var,int,auto,0,level,0

Pass 2 allocates eight words in the current block's stack frame for the specifier and "current information" for the varying string. It compiles the dope into the procedure segment, outside of any executable code sequence. It compiles into the prologue code sequence the code necessary to build the specifier and initialize the string at block entry. It compiles into the epilogue code sequence the code necessary to clear the string.

```
                    ┌──────────────────┐
                    │                  │
To dope in          ├─ ─ ─ ─ ─ ─ ─ ─ ─┤          data pointer
                    │                  │
procedure segment ◄─┼─ ─ ─ ─ ─ ─ ─ ─ ─┤          dope pointer
                    │                  │
To                ◄─┼─ ─ ─ ─ ─ ─ ─ ─ ─┤          free storage pointer
<free_>|[free_]     │                  │
                    ├──────────────────┤
                    │  current offset  │
                    ├──────────────────┤
                    │  current length  │
                    └──────────────────┘
```

Pass 2 compiles, outside of any executable code sequence,

```
        equ             alias,loc           name
.iap:   zero            0,0
        vfd             9/130,27/bits
```

In the prologue code sequence,

```
        eapbp           sp|alias+6,n
        stpbp           sp|alias+0,n
        eapbp           .iap,n
        stpbp           sp|alias+2,n
        eapbp           <free_>|[free_]
        stpbp           sp|alias+4,n
        eax7            alias
        tsx0            .v1
```

And in the epilogue code sequence,

```
        eax7            alias
        tsx0            .v2
```

Loc is the location of the storage assigned to the variable in the stack frame. The symbol .iap is a unique symbol created by Pass 2.

[The instructions above,

```
        eax7            alias
        tsx0            .v1
```

end up calling the procedure varst_$zero. This is a rather expensive way of getting precisely the same effect as,

```
        stz             sp|alias+7
```

which is all that ends up happening. Varst_ is a very
general routine which handles arrays and structures where
it would be a great hardship for the compiler to work
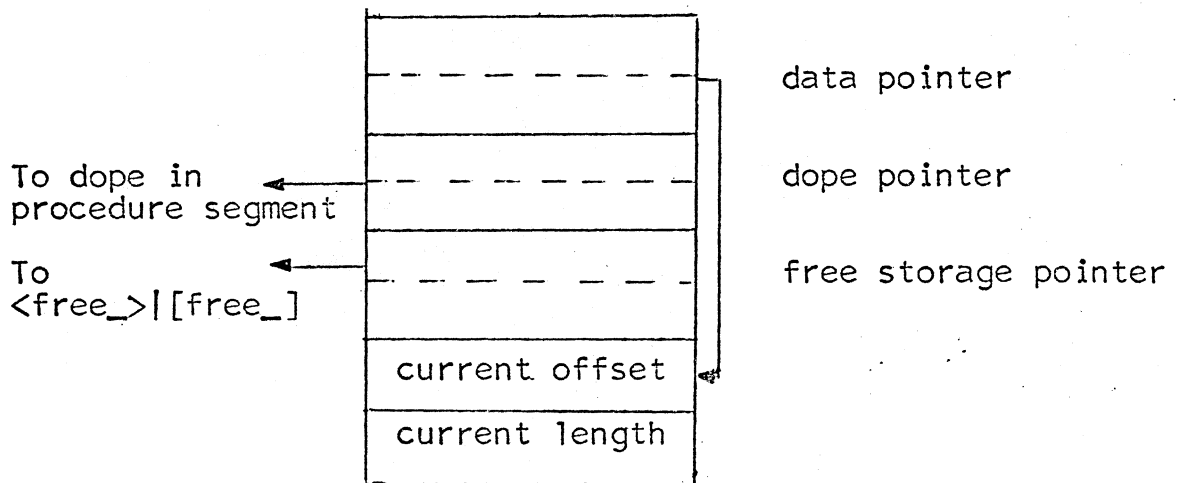these things out. In the present case, however, it is
just silly.]

3.   For an internal static varying string, Pass 1 generates
      the following macro:

```
        dfxx name,alias,bits,0,var,int,stat,0,2,0
```

Pass 2 assigns storage for the specifier and "current
information" for the variable in the procedure's internal
static storage block located at <stat_>|[segname] (for
details see BP.4.00 and BN.5.00.) It compiles the dope
into the procedure segment, outside of any executable
code sequence. It compiles into the "internal static
specifiers" code sequence the code to build the specifier
and initialize the string at first reference.

The string will be laid out in the procedure's internal
static storage in the same fashion as diagrammed above
for automatic strings.

Pass 2 compiles, outside of any executable code sequence,

```
        equ             alias,loc               name
.dvn:   zero            0,0
        vfd             9/130,27/bits
```

And in the "internal static specifiers" code sequence,

```
        eapap           lp|.is,*
        eapbp           ap|0
        adbbp           alias+6,du
        stpbp           ap|alias+0
        eapbp           .dvn
        stpbp           ap|alias+2
        eapbp           <free_>|[free_]
        stpbp           ap|alias+4,n
        eapbp           ap|alias
        stpbp           sp|.u0+2
        tsx0            .v1+2
```

Here <u>loc</u> is the location of the storage assigned to the
variable in the procedure's internal static storage, and
.dv<u>n</u> is a unique symbol created by Pass 2. The symbol
.is is the linkage address of the procedure's internal
static storage, as is mentioned in BN.6.01. The symbols
.u0 and .v1 were discussed earlier.

[As was mentioned above under automatic varying strings,
 the three instructions

```
        eapbp           ap|alias
        stpbp           sp|.u0+2
        tsx0            .v1+2
```

could be relaced by the instruction

```
        stz             ap|alias+7
```

at a considerable saving in time.]

4.   For an <u>external static</u> non-adjustable non-varying string
      without the initial attribute, Pass 1 generates the macro

```
        dfxx name,alias,bits,0,var,ext,stat,0,level,0
```

See BP.4.00 for the implementation of external static
storage. Pass 2 assigns storage for the string specifier
in the procedure's internal static storage. It compiles
the dope and the "trap-before-link" initializer in the
procedure segment outside of any executable code sequence.
It compiles into the "internal static specifiers" code
sequence the code necessary to set up the string specifier.

Pass 2 compiles, outside of any executable code sequence,

```
                equ         alias,loc           name
        .dvn:   zero        0,0
                vfd         9/130,27/bits
        .ym:    dec         2
                dec         1
                arg         *+1
                tsx1        .ei
                eapbp       lp|.is,*
                eapbp       bp|alias
                stpbp       .u0+2
                tsx0        .v1+2
        [       tra         .rt              ]
                segref      stat_,name(datmk_(.ym))
                arg         name
```

And in the "internal static specifiers" code sequence,

```
eapap          lp|.is,*
eapbp          name,n
stpbp          ap|alias+0
eapbp          .dvm
stpbp          ap|alias+2
eapbp          <free_>|[free_]
stpbp          ap|alias+4,n
```

Loc is the location assigned to the specifier in the procedure's
internal static storage. The symbols .dvn and .ym are
unique symbols created by Pass 2. The subroutine .ei
performs an internal procedure save sequence, and is discussed
in BN.6.04. The symbol .rt is the return sequence (discussed
in BN.6.01). The symbols .u0 and .v1 are discussed above.
The symbol .is is the linkage address of the procedure's
internal static storage. The library procedure datmk_
is described in great detail in BP.4.02.

[A bug in EPL at this writing causes the instruction in
square brackets above to be omitted.]

[The astute reader will notice that it is never in fact
necessary to have an initializing procedure attached to
the datmk_ call for an external static item with a specifier.
The same effect can be obtained much more cheaply by putting
the initialization into the "internal static initial"
code sequence.]

[Again it is necessary to point out that the use of .v1 here
is unnecessary and dreadfully inefficient. The instructions

```
eapbp          lp|.is,*
eapbp          bp|alias
stpbp          .u0+2
tsx0           .v1+2
```

could be replaced by

```
eapbp          lp|.is,*
stz            bp|alias+7
```

at a considerable saving in time.]

Non-adjustable Aggregates

See BP.2.01 and BP.2.02 for the details of implementation
of arrays and structures and their dope vectors. All
non-adjustable aggregates of a given storage class are

treated very similarly except in the form of the dope.

When Pass 1 encounters the declaration for a non-adjustable aggregate, it generates a macro of the form,

dfxx name,alias,bits,offset,qv,scope,class,ndim,
        level,nsub

which will be followed by others as described below. Here xx is the data-type: the possibilities in this case are,

pt	pointer
bs	bit-string
cs	character-string
fl	floating
fx	fixed
lb	label
sx	aligned structure
psx	packed structure

The macro fields are discussed in detail in BN.2.02. Briefly: Name is the source-language name of the aggregate. Alias is a unique name created for it by Pass 1. Bits is the precision for arithmetics, the declared length in bits for non-varying strings, the declared maximum length in bits for varying strings, "72" for pointers, "144" for labels, or "0" for structures. Offset is either "0", meaning nothing, or "esi", meaning "external static initial"; see below for a discussion of external static initial data.

Qv is either "xxx", meaning nothing, or "var", meaning varying (if xx is "bs" or "cs"). Scope and class are explained in the discussions of the various storage classes below. Ndim is the number of dimensions for an array ("0" if it is not an array). Level is the block level at which the declaration was encountered. Nsub is the number of substructures (only if xx is "sx" or "psx", meaning the aggregate is an aligned or packed structure or array of structures).

Following the above macro, if ndim is not zero, are "dimension bounds" macros as follows:

dfdb   lower,upper

One of these macros is generated for each dimension of the array. Lower and upper are the bounds for the dimension, and since we are discussing non-adjustable aggregates they are just numbers, like "6".

If the aggregate is a structure, i.e., if xx is "sx" or "psx", then following the dimension bounds macros, if any, come nsub "substructure" macros. They have precisely the same form as the various dfxx macros discussed in this Section and in BN.2.02, with the scope field equal to "mos" (which means "member of structure"), and the "class" field equal to the class field of the major structure macro. If any substructure is non-elementary, i.e., is a structure or array itself, then this entire discussion of Pass 1's actions applies recursively to the substructure.

At this point an example is in order. In an actual compilation, when Pass 1 encountered the following declaration,

```
dcl  1  sigma(7) automatic,
        2 alpha fixed,
        2 beta,
            3 delta float,
            3 eta char(7);
```

it generated the following sequence of macros:

```
dfsx sigma,xx0032,0,0,xxx,int,auto,1,1,2
dfdb 1,7
dffx alpha,xx0033,17,0,xxx,mos,xxxx,0,1,0
dfsx beta,xx0034,0,0,xxx,mos,xxxx,0,1,2
dffl delta,xx0035,27,0,xxx,mos,xxxx,0,2,0
dfcs eta,xx0036,63,0,xxx,mos,xxxx,0,2,0
```

The action of Pass 2 on this example will be discussed later.

1. For a non-adjustable array parameter, the major aggregate macro has scope = "parn", where n is the parameter number, and class = "xxxx".

Pass 2 need do nothing with these macros, but it does in fact generate some harmless equ's which it never uses again and some wasted transfers such as

```
p1.4:      tra              p1.5
```

2. For an automatic non-adjustable aggregate, the major aggregate macro has the class field equal to "auto" and the scope field equal to "int".

Pass 2 assigns space in the current block's stack frame
for the specifier and data of the aggregate. It compiles
the dope into the procedure segment outside of any executable
code sequence. It compiles into the prologue code sequence
the code to generate the specifier at block entry. If
the aggregate contains any varying strings Pass 2 compiles
in the prologue code sequence the code to initialize the
varying strings, and into the epilogue code sequence the
code to clear the varying strings when the block is terminated.

The aggregate is laid out in the stack as follows:
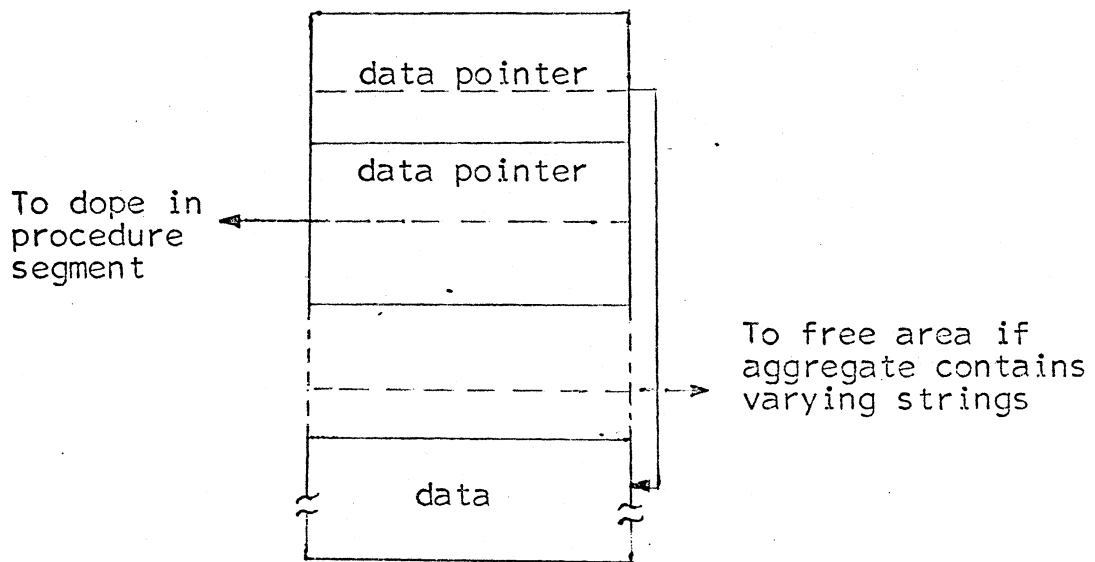
Pass 2 compiles a series of equ's associating the major
aggregate with its stack location and each substructure
with its substructure number. For the major aggregate:

        equ alias,loc   name

and for each subaggregate:

        equ alias,subno name

Outside of any executable code sequence:

        .ian: ...   (dope: see BP.2.02)

In the prologue code sequence, if the aggregate contains no varying strings:

```
        eapbp           sp|alias+4,n
        stpbp           sp|alias+0,n
        eapbp           .ian,n
        stpbp           sp|alias+2,n
```

In the prologue code sequence, if the aggregate does contain varying strings:

```
        eapbp           sp|alias+6,n
        stpbp           sp|alias+0,n
        eapbp           .ian,n
        stpbp           sp|alias+2,n
        eapbp           <free_>|[free_]
        stpbp           sp|alias+4,n
        eax1            alias
        tsx0            .v2
```

In the epilogue code sequence, if the aggregate contains varying strings:

```
        eax1            alias
        tsx0            .v2
```

Here loc is the location assigned to the aggregate in the stack frame, subno is the number of substructure within its immediate containing structure, and .ian is a unique symbol generated by Pass 2.

For the example aggregate shown above, Pass 2 generates:

```
          equ           xx0032,36 sigma
          equ           xx0033,1  alpha
          equ           xx0034,2  beta
          equ           xx0035,1  delta
          equ           xx0036,2  eta
.ia0:     zero          -4,0
          zero          320*512,2
          dec           28
          zero          0,1
          zero          .ia0+10-*,256
          zero          320*512,1
          dec           28
          dec           4
          dec           1
          dec           7
          zero          1,0
          zero          256*512,2
```

```
                    dec         3
                    zero        0,1
                    zero        .ia0+15-*,128
                    zero        1,0
                    vfd         9/128,27/63
        pl.1:       eapbp       sp|xx0032+4,n
                    stpbp       sp|xx0032+0,n
                    eapbp       .ia0,n
                    stpbp       sp|xx0032+2,n
```
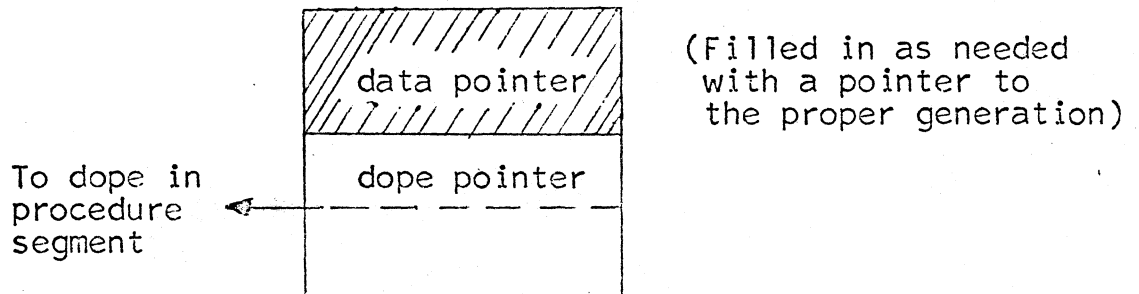
3.   For a <u>controlled</u>, <u>based</u> non-adjustable aggregate (which by
      definition cannot contain varying strings,) the major
      aggregate macro has <u>scope</u> = "int" and <u>class</u> = "cont".

Pass 2 allocates space in the current block's stack frame
for a specifier for the aggregate. It compiles the dope
into the procedure segment, outside of any executable
code sequence.   It compiles into the prologue code sequence
the code to fill in the "dope pointer" in the specifier.
The "data pointer" in the specifier is filled in whenever
a generation of the string is accessed: see BN.6.03.

Thus what will be in the stack will look like:

```
                        ////////////////
                        / data pointer //        (Filled in as needed
                        ////////////////          with a pointer to
                                                  the proper generation)
To dope in              ┌──────────────┐
procedure     ◁─────────┤ dope pointer │
segment                 ├ ── ── ── ── ─┤
                        │              │
                        └──────────────┘
```

Pass 2 compiles, as usual, the <u>equ</u>'s associating the alias
for the major aggregate with the stack location for its
specifier and the alias for each substructure with its
substructure number. It compiles, outside of any executable
code sequence,

        .ia<u>n</u>:   ...   (dope: see BP.2.02)

And in the prologue code sequence,

```
            eapbp               .ia<u>n</u>
            stpbp               <u>alias</u>+2,n
```

Here as always .ia<u>n</u> is a unique symbol generated by Pass 2.

4.   For an <u>internal</u> <u>static</u> aggregate, the major aggregate
      macro has <u>scope</u> = "int" and <u>class</u> = "stat".

Pass 2 assigns storage for the specifier and data of the
aggregate in the procedure's internal static storage.
It compiles the dope into the procedure segment, outside
of any executable code sequence. It compiles into the
"external static specifiers" code sequence the code to
build the specifier.

The aggregate will be laid out as diagrammed earlier for
automatic non-adjustable aggregates.

Pass 2 generates the <u>equ</u>'s necessary to associate the
alias of the major aggregate with the storage assigned
to it in internal static storage and to associate the
alias of each substructure with its substructure number.

It compiles, outside of any executable code sequence,

.dv<u>n</u>:   ...   (dope: see BP.2.02)

And in the "internal static specifiers" code sequence,
if the aggregate does not contain any varying strings,

```
eapap          lp|.is,*
eapap          ap|0
adbbp          alias+4,du
stpbp          ap|alias+0
eapbp          .dvn
stpbp          ap|alias+2
```

Or if the aggregate does contain varying strings,

```
eapap          lp|.is,*
eapbp          ap|0
adbbp          alias+6,dn
stpbp          ap|alias+0
espbp          .dvn
stpbp          ap|alias+2
eapbp          <free_>|[free_]
stpbp          ap|alias+4,n
eapbp          ap|alias
stpbp          sp|.u0+2
tsx0           .v1+2
```

The symbol .is is the linkage address of the procedure's
internal static storage. The symbol .dv<u>n</u> is a unique
symbol generated by Pass 2.

5. For an <u>external static</u> aggregate, the major aggregate macro
    has <u>scope</u> = "ext" and <u>class</u> = "stat".

Pass 2 allocates storage for the specifier of the aggregate
in the procedure's internal static storage. It sets up
a trap-before-link out-reference which will cause the
procedure datmk_ to "grow" the storage needed on first
reference. If the aggregate contains varying strings,
the call to datmk_ includes an initializer to initialize
those varying strings. It compiles the dope into the
procedure segment, outside any executable code sequence.
It compiles into the "internal static specifiers" code
sequence the code to create the specifier.

Pass 2 compiles the necessary <u>equ</u>'s to associate the alias
for the major aggregate with the location of its specifier
in the procedure's internal static storage, and to associate
the alias for each substructure with its substructure
number. If the aggregate contains no varying strings
it compiles, outside any executable code sequence,

```
.dvn:      ...            (dope:  see BP.2.02)

.ym:       dec      ·   nwords
           dec          0
           segref       stat_,name(datmk_(ym))
           arg          name
```

Again if the aggregate contains no varying strings, Pass
2 compiles into the "internal static specifiers" code
sequence,

```
           eapap        lp|.is,*
           eapbp        name,n
           stpbp        ap|alias+0
           eapbp        .dvn
           stpbp        ap|alias+2
```

If on the other hand the aggregate contains varying strings, Pass
2 compiles, outside of any executable code sequence,

```
.dvn:      ...            (dope:  see BP.2.02)

.ym:       dec          nwords
           dec          1
           arg          *+1
           tsx1         .ei
           eapbp        lp|.is,*
           eapbp        bp|alias
           stpbp        sp|.u0+2
           tsx0         .v1+2
           tra          .rt
```

And in the "internal static specifiers" code sequence,

```
eapap        lp|.is,*
eapbp        name,n
stpbp        ap|alias+0
eapbp        .dvn
stpbp        ap|alias+2
eapbp        <free_>|[free_]
stpbp        ap|alias+4,n
```

Note that .is is the linkage address of the procedure's internal static storage, and .dvn and .ym are unique symbols created by Pass 2.

Adjustable Items

See the preceding discussions of non-adjustable strings and aggregates for a general outline of what the macros for items with specifiers look like. In the parlance of the EPL project, an "extent" is a number which is either an array bound or a string length. This concept is very useful in what follows.

For each adjustable extent in an item, Pass 1 generates the macros for a subroutine which evaluates the expression for the extent. Then in the dfxx macro for the item it makes the offset field equal to "adj" and puts the name of the appropriate subroutine in any place where a number (like "6") for the extent would have appeared if the extent were not adjustable.

See BN.6.03 for a discussion of expression-evaluation. The subroutine which Pass 1 generates to evaluate an extent has the form

```
use          contbds
dclb         , extentalias,144,0,xxx,con,xxxx,0,level,0
...          (macros to evaluate expression and end up
              with a 17-bit integer in the "accumulator")
that"s all
use main
```

The use macro controls code sequences; see BN.6.01 for exactly how code sequences are handled in Pass 2. Extentalias is a unique name generated by Pass 1 for the extent-calculating subroutine. The macro that"s all specifies a return from the extent subroutine.

Pass 2 treats the extent subroutine half as a separate
block and half as an ordinary code sequence. The extent
subroutine will be called using the rather peculiar calling
sequences described later in the discussions of the various
storage classes. These calling sequences involve a "push"
of the call stack, the creation of a display (displays
are described in BN.6.04), and a tsx2 instruction. The
extent subroutine thus operates at a block level one higher
than the block level of the declaration. (The fiddling
with levels allows non-local use of a based adjustable
item.) It evaluates the extent expression and leaves
it as an integer in the $\underline{a}$ - register. The code takes
the following form:

```
extentalias:   null"
               ...        (expression evaluator)
               tra        0,2
               equ        .un,...
               equ        .asn,...
               equ        .wn,...
```

The equ's at the end are standard equivalences always
generated at the end of block. See BN.6.04.

It is a fortunate fact about the design of PL/I - EPL
dope vectors in Multics that any extent goes into exactly
one place in the dope vector, right-adjusted in the word.
This fact is of some importance in the discussions later
of the various storage classes.

An example: in an actual compilation, when Pass 1 encountered
the declaration,

```
dcl 1 sigma (n) automatic,
      2 alpha fixed,
      2 beta,
            3 delta float,
            3 eta char (2*n);
```

where $\underline{n}$ had been declared in an outermore block and had
alias xx0026, it generated the following sequence of macros:

```
use        contbds
dclb       ,xx0031,144,0,xxx,con,xxxx,0,2,0
dffx       ,xx0032,17,0,xxx,int,auto,0,2,0
ldfx       xx0026,17,0,xxx,int,auto,0,1,0
stfx       xx0032,17,0,xxx,int,auto,0,2,0
ldfx       xx0032,17,0,xxx,int,auto,0,2,0
that"s all
```
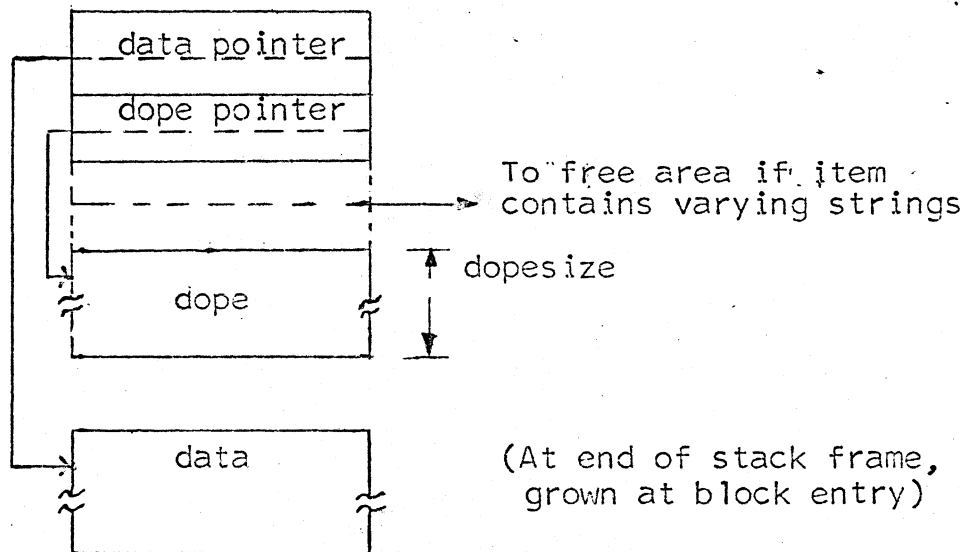
```
use        main
use        contbds
dclb       ,xx0038,144,0,xxx,con,xxxx,0,2,0
dcfx       2,xx0040,4,0,xxx,con,xxxx,0,1,0
dffx       ,xx0039,17,0,xxx,int,auto,0,2,0
ldfx       xx0040,4,0,xxx,con,xxxx,0,1,0
fxfx       22,0,17,0,
stfx       xx0039,17,0,xxx,int,auto,0,2,0
ldfx       xx0039,17,0,xxx,int,auto,0,2,0
that"s all
use        main
dfsx       sigma,xx0029,0,adj,xxx,int,auto,1,2,2
dfdb       1,xx0031
dffx       alpha,xx0034,17,0,xxx,mox,xxxx,0,1,0
dfsx       beta,xx0035,0,adj,xxx,mos,xxxx,0,1,2
dffl       delta,xx0036,27,0,xxx,mos,xxxx,0,2,0
dfcs       eta,xx0037,xx0038,adj,xxx,mos,xxxx,0,2,0
```

This example is taken up again below in the discussion of the
automatic storage class.

1.   For an automatic adjustable item, Pass 2 allocates
      sufficient space in the current stack frame for the
      specifier and dope of the item. It compiles in the
      prologue code sequence the code which both creates the
      dope and specifier and grows the stack frame sufficiently
      to hold the data. The code to create the dope vector
      copies a "template" dope vector into the stack frame, calls
      the various extent subroutines and stores their results
      into the proper places in the dope vector, and finally
      calls the run-time procedure tdope_ (described in BN.7.01)
      to fill in missing details (such as offsets) and return
      the number of words required.

The aggregate will end up laid out in the stack frame as follows:

data pointer

dope pointer

To free area if item contains varying strings

dopesize

dope

data

(At end of stack frame, grown at block entry)

Pass 2 compiles the usual set of equ's associating the
item's alias with the location in the stack frame of its
specifier and each substructure's alias with its substructure
number. (These equ's were described above under Non-
Adjustable Aggregates.) It compiles, outside any executable
code sequence,

        .iap: ...   (dope vector template)

and may things in the prologue sequence; first the code
to copy the template dope vector into the stack and create
the specifier:

```
        ldx2      dopesize-1,du
        lda       .iap,2
        sta       sp|alias+4,2    (+6 if varying strings are involved)
        sblx2     1,du
        tpl       *-3
        eapbp     sp|alias+4      (+6 if varying strings are involved)
        stpbp     sp|alias+2
        eapbp     sp|18,*
        stpbp     sp|alias
```

Then the code to push the call stack and create a display:

```
        eax7      .mn
        tsx0      .sv
        eapbp     sp|16,*
        stpbp     sp|.ds
        eapbp     sp|.ds,*
```

```
    [   ldaq      bp|.ds      ]
        staq      sp|.ds+2    ]        move a short display

    [   eax4      nrc         ]
        tsx0      .cp         ]        move a long display
```

(One of the two pieces of code in brackets above is chosen
depending upon the level of the declaration. See below.)

Then the code to put the address of the dope vector into a
known place in the new stack frame:

```
        eapbp     sp|.ds,*
        eapbp     bp|alias+4    (+6 if varying strings are involved)
        stpbp     sp|.wn
```

Then for each adjustable extent the code to call the extent
subroutine for that extent and store its result into the
proper word of the dope:

```
        tsx2        extentalias

[    lrs        36       ]       Needed only for
     mpy        9,du     ]       the length of a
     lls        18       ]       character-string.
     .
[    ora        idcode*512,du] Not needed for an array bound

     eax4       dopeword
     sta        sp|.wn,*4
```

Then the code to call an interface subroutine which calls
tdope_ and brings the call stack level back down:

```
     eapbp       sp|.wm,*
     tsx0        .dp1
```

Finally, if varying strings are involved, the code to
call varst_$zero to initialize the varying string to zero
length:

```
     eax7        alias
     tsx0        .v1
```

This ends the code generated in the prologue code sequence.

If varying strings are involved, the following code is
compiled into the epilogue code sequence to clear the
varying strings at block termination:

```
     eax7        alias
     tsx0        .v1
```

[The notes given here on the case where varying strings
are involved may well turn out to be wrong. A bug in
EPL at this writing causes the case of adjustable items
containing varying strings to be compiled into nonsense.]

In the code shown above, .iap is a unique symbol created
by Pass 2. Dopesize is the size of the dope vector in
words. The symbol .ds is the stack location of the "display"
(see BN.6.04). The symbols .mn and .wn are special symbols
defined for the current block. The .mn gives the maximum
stack frame size needed by any dope-building code in the
block. The .wn is a stack location designated as the

place in a dope-building stack frame where the location
of dope is to be stored. Idcode is the id code of a string,
if it is needed in the dope word (see BP.2.02 for id codes.).
The subroutines .v1 and .v2 were described earlier (under
Non-adjustable Varying Strings). The number nrc is the
member of its pairs to be moved into the display. The
subroutine .cp is described in BN.6.04.

The subroutine .dp1 is compiled by Pass 2 into every procedure
which needs it. It does several things: (1) calls the
library procedure tdope_ (see BN.7.01) to fill in missing
pieces of the dope vector; (2) pops the call stack back
where it belongs; (3) increases the size of the current
stack frame sufficiently to hold the data for the item.
The code for .dp1 is:

```
.dp1:       stpbp       sp|.u0
            eapbp       sp|.u0+2 .
            stpbp       sp|.u0+2
            call        <tdope_>|[tdope_] (sp.|.u0-2)
            ldq         sp|.u0+2
            adq         7,du
            anq         -8,du
            eapsp       sp|16,*
            asq         sp|19
            tra         0,0
```

Note that .u0 is a "utility" space set aside in every
stack frame. It is used here and in many other places.
[The call to tdope_ is highly non-standard and is unacceptable
as it stands.]

[Again we have a chance of overflow in the instruction

```
            adq         7,du
```

which should be replaced by

```
            eaq         7,qu
```

and in the instruction

```
            asq         sp|19
```

which should be replaced by

```
            eapbp       sp|18,*qu
            stpbp       sp|18
```

here again the sharp eyes of C. G. Garman are responsible
for the discovery of this bug.]

The Example: see the example of an adjustable aggregate
given earlier. When Pass 2 encountered the collection
of macros which Pass 1 had generated, it compiled:

```
                tra             s2.1
    .b0:xx0031:                 null        "
                equ             xx0032,38
                tra             .b1
    .b1:        eapbp           sp|.ds+2,*
                lda             bp|xx0026
                eapbp           sp|.ds+0,*
                sta             bp|xx0032
                eapbp           sp|.ds+0,*
                lda             bp|xx0032
                tra             0,2
                equ             .u3,42
                equ             .as3,56
                equ             .w3,42
    .b2:xx0038:                 null        "
                tra             .b3
    xx0040:     dec             2
                equ             xx0039,39
    .b3:        lda             xx0040
                lrs             36
                eapbp           sp|.ds+2,*
                mpy             bp|xx0026
                lls             36
                eapbp           sp|.ds+0,*
                sta             bp|xx0039
                eapbp           sp|.ds+0,*
                lda             bp|xx0039
                tra             0,2
                equ             .u4,42
                equ             .as4,56
                equ             .w4,42
                null
                equ             xx0029,40        sigma
                equ             xx0034,1         alpha
                equ             xx0035,2         beta
                equ             xx0036,1         delta
                equ             xx0037,2         eta
    .ia0        zero            -2,0
                zero            320*512,2
                dec             0
                zero            0,1
                zero            .ia0+10-*,256
                zero            320*512,1
```

```
              dec            0
              dec            2
              dec            1
              dec            0
              zero           1,0              xx0031
              zero           256*512,2
              dec            1
              zero           0,1
              zero           .ia0+15-*,128
              zero           1,0
              vfd            9/128,27/
p2.1:         ldx2           17-1,du
              lda            .ia0,2
              sta            sp|xx0029+4,2
              sblx2          1,du
              tpl            *-3
              eapbp          sp|xx0029+4
              stpbp          sp|xx0029+2
              eapbp          sp|18,*
              stpbp          sp|xx0029
              eax7           .m2
              tsx0           .sv
              eapbp          sp|16,*
              stpbp          sp|.ds
              eapbp          sp|.ds,*
              ldaq           sp|.ds
              staq           sp|.ds+2
              eapbp          sp|.ds,*
              eapbp          bp|xx0029+4
              stpbp          sp|.w2
              tsx2           xx0031
              eax4           9
              sta            sp|.w2,*4
              tsx2           xx0038
              lrs            36
              mpy            9,du
              lls            18
              ora            128*512,du
              eax4           16
              sta            sp|.w2,*4
              eapbp          sp|.w2,*
              tsx0           .dp1
              tra            p2.2
```

2.   For a <u>controlled</u>, <u>based</u> adjustable string or aggregate,
      the code compiled is even more inefficient than that for
      an automatic adjustable item. The action which must
      take place at block entry to set up the dope must here
      take place instead at every reference to the item.

[Since based adjustable structures appear throughout the
Multics system, this would seem to be the area in which
optimization is most worthwhile. It is also the area
in which optimization is most difficult, however.]

Pass 2 allocates space for the item's specifier and dope
in the stack frame of the current block. It compiles
into the prologue code sequence the code to copy a template
dope vector into the stack and to create half a specifier
(the other half is filled in when a generation of the
based item is accessed). It compiles an internal procedure
to do the extent calculation and set up the dope properly.
This internal procedure must be called (using the standard
internal procedure call sequence, see BD.7.02) just before
every reference to the item.

The specifier and dope will be laid out in the stack as follows:
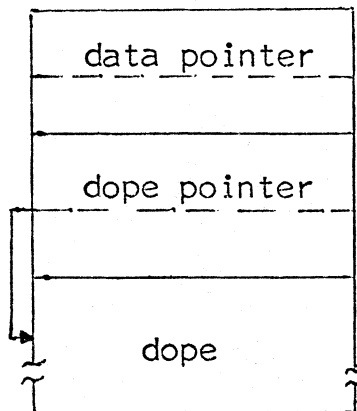
(To be filled in as needed with
 a pointer to the proper
 generation)

Pass 2 compiles the <u>equ</u>'s which associate the various
aliases with the right numbers. (See the discussion of
these <u>equ</u>'s earlier under <u>Non-adjustable Aggregates</u>.)
It compiles, outside of any executable code sequence,

      .iap: ...   (dope vector template)

And in the prologue code sequence,

```
        ldx2        dopesize-1,du
        lda         .iap,2
        sta         sp|alias+4,2
        sblx2       1,du
        tpl         *-3
        eapbp       sp|alias+4
        stpbp       sp|alias+2
```

And outside of any executable code sequence, the internal procedure to call the extent subroutines and create the dope. This procedure starts,

```
.ctm:       eax7        .mn
            tsx0        .sv
            tsx0        .cp
```

It creates a display and calls the extent subroutines using precisely the same code as the prologue code for automatic adjustable items. Finally it goes to a common subroutine which calls tdope_ and does a return:

```
        eapbp       sp|.wn,*
        tra         .dp0
```

See the earlier discussion of automatic adjustable items for definitions of most of the terms used above. The subroutine .cp is part of the save sequence for an internal procedure. It is described in BN.6.04.

The subroutine .dp0 is compiled by Pass 2 into every procedure which needs it. It calls the library procedure tdope_ to fill in the details of the dope vector and then does a return. The code is always

```
.dp0:       stpbp       sp|.u0
            eapbp       sp|.u0+2
            stpbp       sp|.u0+2
            call        <tdope_>|[tdope_] (sp|.u0-2)
            tra         .rt
```

Constants

When Pass 1 encounters a fixed, floating, bit-string, or character-string constant, it generates a macro of the form,

```
dcxx text,alias,bits,0,xxx,con,xxxx,0,1,0
```

Here xx is the type:

    fx   fixed
    fl   float
    bs   bit-string
    cs   character-string

The macro fields are described in BN.2.02: briefly text
is the text of the constant exactly as it appeared in
the source program, alias is Pass 1's unique name for
the constant, and bits is the precision of an arithmetic
constant or the length in bits of a string.

1.   For a fixed constant with precision ≤ 35, Pass 2 compiles,
      outside of any executable code sequence,

          alias: dec   text

      and if precision ≥ 36,

                      even
          alias: dec   textb71d

2.   For a floating constant of any precision, Pass 2 compiles,
      outside of any executable code sequence,

                      even
          alias: dec   textm

      where textm is text modified by replacing the required
      "e" in the constant with a "d".

Note that because of the way single- and double- precision
floating-point is handled in the 645, the compiler may
assume that every floating-point number is double-precision.
[This streamlining costs 1 1/2 words per single-precision
floating constant: cheap at half the price.]

3.   For a bit constant, Pass 2 compiles the following code,
      outside of any executable code sequence:

          arg   *+5
          arg
          arg   *+1
          zero
          vfd   9/160,27/bits
          vfd   ...

where the ellipsis represents a very strange variable
field for the vfd which ends up putting the proper bit-pattern
left-justified in a block of as many words as are needed.

The first three words, if accessed only as indirect words,
are a specifier that can live in a pure procedure. The
next two words are dope.

4.   For a <u>character</u> constant, Pass 2 compiles, outside of
       any executable code sequence,

```
       arg         *+5
       arg
       arg         *+1
       zero
       vfd         9/160,27/bits
       aci         text
```

The <u>aci</u> pseudo-op is very convenient since it treats its
argument precisely as a PL/I - EPL quotation.

<u>Label Prefixes</u>

When Pass 1 encounters a label prefix, it generates the macro,

```
       dclb        name,alias,144,0,xxx,con,xxxx,0,1,0
```

where <u>name</u> is the name in the source program and <u>alias</u>
is Pass 1's unique name for it.

Pass 2 compiles an eplbsa label prefix from this macro:

```
       alias:
```

Other Pass 2 - generated code may then appear on the same line.

<u>The Null Pointer</u>

When Pass 1 encounters a reference to the built-in function
<u>null</u>, it generates the macro,

```
       dcpt        null,xx0000,72,0,xxx,con,xxxx,0,0,0
```

The <u>dcpt</u> macro is a constant: the fields never take on
any other values.

Pass 2 compiles, outside of any executable code sequence,

```
               even
       xx0000:  its         -1,1,n
```

Variables with the Initial Attribute

When Pass 1 encounters the declaration of a variable with the initial attribute, it compiles into the appropriate code sequence the code for an assignment of the initial value to the variable. Which code stream is "appropriate" depends upon the storage class of the variable; this issue is the primary concern of the discussions which follow.

Given the declaration

        dcl   a   ...   initial (b);

the macros generated by Pass 1 to do the initial assignment are identical to those for the assignment statement

        a = b;

See BN.6.03 and BN.6.08 for details of expression evaluation and the assignment statement.

The dfxx macro which Pass 1 generates for the variable involved has precisely the form shown in the earlier discussions of the various kinds of variables, with one exception noted below in the discussion of external static initial data. Pass 2 takes the same action on seeing these macros.

1.   For an automatic variable with the initial attribute, the initializing macros generated by Pass 1 have the form

        use   autoinitint,alias
        ...   (initializing macros)
        use   main

The use macros are Pass 1's control of code sequences: see BN.6.01 for further discussion. Pass 2 simply compiles into the "automatic initial" code sequence the normal code it would compile for the initializing macros.

2.   For a controlled, based variable with the initial attribute, the initializing macros generated by Pass·1 take the form:

        use   continitint,alias
        ...   (initializing macros)
        use   main

[At this writing Pass 2 does not recognize the first macro
shown above, and consequently controlled initial does
not work. Presumably what should happen is that Pass
2 compiles the sequence of initializing macros as a `tsx`-able
subroutine called by the code for the allocate statement.]

3.   For an _internal_ _static_ variable with the initial attribute,
      the initializing macros generated by Pass 1 have the form,

              use   statinitint,_alias_
              ...   (initializing macros)
              use   main

Pass 2 compiles the usual code for the initializing macros
in the "internal static initial" code sequence. See BN.6.03
for expression evaluation and BN.6.01 for code sequences.

4.   For an _external_ _static_ variable with the initial attribute,
      Pass 1 generates initializing macros in the following form:

              use   statinitext,_alias_
              ...   (initializing macros)
              use   main

And then generates the macro defining the variable precisely
as described earlier but with the _offset_ macro field equal
to "esi".

Pass 2 compiles code, outside of any executable code sequence,
in the following form:

        init._alias_: ...   (initializing code)
                           tra   .rt
          ._yp_: dec   _nwords_
                           dec   1
                           arg   *+1
                           tsx1   .ei
                           tra   .init._alias_
                           segref   stat_,_name_(datmk_(._yp_))
                           link   _alias_,_name_

If the variable requires a specifier the code to create
it is compiled into the "internal static specifiers" code
sequence. If the variable is a varying string, the code
above will also include a call to _.v1_ to initialize the
string.

The subroutines .rt and .ei, and .v1 are described elsewhere.
Nwords is the number of words which must be grown by datmk_
for the variable.

Members of Structures with the Initial Attribute

When Pass 1 encounters a member of a structure with the
initial attribute, the initializing macros it produces
take the form,

```
    use         xxxxinitmos
    ...         (initializing macros)
    use         main
```

for any storage class at all.

Unfortunately at this writing Pass 2 compiles utter nonsense
for this sequence of macros.

Label Arrays with the Initial Attribute

EPL does not in general allow initialized arrays; however
it does allow a peculiar version of the initial attribute
for label arrays which gives a more-or-less natural way
of programming a many-way fork on an integer variable.
See the documentation [which does not exist] of this language
feature for details of its use.

The implementation is best shown through an example.
In an actual compilation, when Pass 1 encountered the
statements

```
    dcl fork  (5)  label initial(a,b,c,d,e);
    a:b:c:d:e:;
```

(Clearly a rather artificial example), Pass 1 generated
the following macros:

```
    use         xxxxinitcon,xx0026
    golb        xx0030,144,0,xxx,con,xxxx,0,1,0
    golb        xx0031,144,0,xxx,con,xxxx,0,1,0
    golb        x x0032,144,0,xxx,con,xxxx,0,1,0
    golb        xx0033,144,0,xxx,con,xxxx,0,1,0
    golb        xx0034,144,0,xxx,con,xxxx,0,1,0
    use         main
    dclb        fork,xx0026,144,0,xxx,con,xxxx,1,1,0
    dfdb        1,5

    dclb        a,xx0030,144,0,xxx,con,xxxx,0,1,0
    dclb        b,xx0031,144,0,xxx,con,xxxx,0,1,0
    dclb        c,xx0032,144,0,xxx,con,xxxx,0,1,0
    dclb        d ,xx0033,144,0,xxx,con,xxxx,0,1,0
    dclb        e,xx0034,144,0,xxx,con,xxxx,0,1,0
```

And Pass 2 then compiled the following code:

```
                tra         s1.1
init.xx0026:                tra         xx0030
                tra         x x0031
                tra         xx0032
                tra          xx0033
                tra         x x0034
.ia0:           zero        -1
                zero        65*512,1
                dec         5
                dec         1
                dec         1
                dec         5
p1.1:           tra         p1.2
xx0026:         arg         init.xx0026              "fork
                arg
                arg         .ia0
"
s1.1:xx0030:                null        "a
xx0031:         null        "b
xx0032:         null        "c
xx0033:         null        "d
xx0034:         null        "e
```