Description of the object program

The main points seem to be:

1. Memory lay out
2. Simple variable adressing
3. Block entry and exit
4. Formal variable addressing
5. Procedure calls and actual parameters
6. Subscription
7. Labels and switches (goto-statement)
8. Conditional expressions and statements
9. For-statement
10. Library
11. Assignment
12. Error checking
13. Input-output buffering.

Memory Layout

Apart from a fixed ~~point~~ portion of core memory, the remainder will be used in portions of 512 words, called a core page.

Memory on drum will be divided into drum pages of 512 words. The starting address is given by

$$1025 \cdot \text{drum page number} \pmod{2^{19}}$$

The information in a program is subdivided into segments also of 512 words. (A segment is an information unit, a page is a memory unit)

In the case of a big array, a segment which has been declared, but up till now is still unused, is regarded as existing, but empty. It occupies no page.

The information of a non-empty segment may be at any moment
1) in the process of transportation
2) exclusively on core
3) exclusively on drum
4) on both.

Remark: while a segment is in the process of transportation it may cease to exist. As far as the running system is concerned the core page remains occupied until notice has been taken of the completion of the transport process.

We distinguish (mainly)
- stock segments
- program segments
- array segments.

Remark 1. Each program has a unique stack.

2. Program segments are in principle of two kinds, library segments and private segments ("private" will mean: pertaining to an individual program).

3. Array segment may turn out to be different from buffer segments for input and output. Time will show.

To each segment belongs a so-called SV (Segment Variable); it is via the SV that a segment will be addressed.

Addressing Program Segments

In core memory will be a table of SV's pertaining to the library segments. This will contain as many SV's as the library has segments.

The SV table for the library will be situated at a fixed place in store; the SV's will be addressed - mirabile dictu - by their physical address.

The private program SV's of a program will be put at the bottom of its stack. This bottom page will be chosen, fixed, at the beginning of the translation process. As a result the private program SV's can be addressed by their physical address.

Points in program or library are thus characterized by 9 bits line number + 18 (or may be 15, undecided yet) bits SV-address.

Addressing Array Segments

An array will be local to a program, the SV's corresponding to its segments will be put in the stack at a block height corresponding to the declaration. They are never addressed directly by the object program.

Addressing buffer segments

Undecided yet.

Remark about stack segments

A stack segment will always be in core, and on a so-called "holy core page". Therefore, the system can safely convert to physical addresses when referring to elements within the stack.

Stack segment overflow

Logically consecutive stack segments need not be on consecutive core pages. The object program has the duty to reserve, at block entry, "so and so many more" stack places. Whether they can be put on the same segment or whether the transition to a next page is needed is not reflected in the object program.

Simple variable addressing

Reading - i.e. executing - a text implies that D points to a proper display.

Quantities declared in the outermost block will be addressed by M0[q]-addresses, simple variables in blocks with block height 1 by M1[q] addresses, etc.

However: when executing a piece of text of block height n. D[n] should be contained in the A-register, so that for local simple variables the addressing can also be done by means of MA[q]. For the system this implies, that after return from a primitive to program text, care must be taken to ensure that A will indeed contain the local reference point. (At each stage D[-1] = n, so this is one of the ways to achieve this.)

Simple reals will occupy two words in the stock, simple booleans and integers will occupy a single word. Simple complex variables will contain four consecutive words in the stack.

The local variables of a block will be addressed by Mn[q] (or MA[q], see above) with q = 0,1,2,3... n with n $\leq$ 255. The rule that we start with q = 0 is essential; whether the maximum value of n will be allowed to be 255 is a translator question: it may decide to introduce a fictitious extra block at an earlier point.

Block entry

The main task of block entry is
    a) extension of the terminology (=Display)
    b) stack reservation.
The block entry apart from arrays will be performed in the following steps.

First a stack reservation has to be made for the explicitly addressed stack quantities (one word per simple boolean, and simple integer, two words per simple real, four words per simple complex and one word per array). To these have to be added some standard ones describing the linking of stack storage; see below.

The stack reservation is made by two instructions:
    S:= number of words consecutively needed
    SEO
Between these two orders, segment transition will not be allowed.
SEO is the name of a subroutine jump to a System Routine. It will be an instruction of the type SUBCD.

The task of SEO is to increase the current block number, (to be found in D[-1]), to find the new stack space asked for, either in the same or in another stack page, to provide the linking necessary, to set the WP, the AP and to fill in a new item in the Display.

When no array is declared, the object program will proceed by asking for an anonymous reservation, i.e. the amount of storage space needed for the anonymous intermediate results. The algorithm describing how to derive this number will be given in detail.

In the object program the anonymous reservation will be described by

S:= number of words consecutively needed
SE1

Also SE1 is a SUBCD instruction; between these two orders no segment transition will be allowed.

The task of SE1 is to check whether WP can remain as it stands, or whether it must be set to the beginning of a new stack page.

In the case of one or more array declarations the proceedings are slightly different.

After SE0, which effected the block entry proper the array declarations are treated "array segment" after "array segment" (see 5.2.1.).

The treatment of an array segment consists of

1) asking stack space for the values of the bounds in order from left to right and their anonymous evaluation. (In this respect the creation of storage mapping functions, the processing of an "array segment" can be regarded as a 2n-ary operation).
2) evaluating the bound values and stacking them as integers, one word for each, on top of the stack (i.e. at the bottom of the place reserved). These values will remain there during the execution of the block.
3) creation of the storage function. This operation needs as parameters
3.1) the type of the array (boolean,integer,real,or complex)
3.2) the bound values
3.3) the dimension
3.4) the array words concerned.

The type of the array will be specified by different entries.
The bound values of the array can be found at the top of the stack.
The dimension can be deduced from the difference between AP and WP.
The array words will be specified by the object program,
as follows:

S:= number of array identifiers in the segment.
A:= physical address of the first array word concerned.

These two instructions will be followed by one of the following four:

SE2	create integer arrays	} of type SUBCD.
SE3	create real arrays	
SE4	create boolean arrays	
SE5	create complex arrays	

The return value of A is the local reference point, of B the present AP, of S and F immaterial.

The three instructions thus generated may not be separated by a segment transition.

The first array segment is now treated. The object program will repeat this for any further array segments; after the last one, SE1 will be used to reserve stack space for the anonymous expression evaluation. This is needed, because SE2, SE3, SE4 and SE5 will imply a (statically unknown) increase of WP.

The ********** procedures declared within a block will not result in extra operations in the object program, to be performed at block entry. The processing of switch declarations is undecided yet.

Block exit

The normal block exit ("passing through an end") will be reflected in the object program by a single instruction of type SUBCD

SE6: Block exit.

Formal variable addressing

The top of the stack is used as transmission mechanism to transmit the actual parameters to the procedure called.

At call side the object program stacks the current actual parameters on top of the stack in some prescribed order (either from left to right or right to left; this is fully a translator question and need ** not be decided now). Each actual parameter specification takes four consecutive words in the anonymous top. As soon as the procedure has been duly entered, they become "nonymous" and can be addressed as the so-called "formal locations".

The top formal locations are addressable by something like Mn[-4] up to and including Mn[-1], the next formal locations by Mn[-8] up to Mn[-5] etc. with suitable n,mx viz. one higher than the block height of the local variables of the block in which the procedure has been declared.

Note. If the number of local variable exceeds the limit, so that one or more additional block parenthesis pair has to be introduced by the translator, then the procedures and switches declared within this block must be treated as declared within the innermost block.

The processing of the specifications and the value list will be described as part of the procedure entry. In this section we shall restrict mm ourselves to formals outside the value list.

Here the translator will not distinguish between the specifications integer and real; to indicate this we shall use the term "arithmetic".

The processing of arithmetic formals

The formal locations Mn[-4*m-alpha+0] ... Mn[-4*m-alpha+3] be denoted as f[0] ... f[3]. (alpha leaving space for return information.)

Essentially, these formal locations will be processed in three different ways:

a) as actual parameter
b) as right hand value
c) as left hand value.

Actual parameter

The structure of the object program transmitting a formal variable as an actual one will be described as part of the calling sequence. The contents of the formal locations have to be copied on top of the stack.

Right hand value

The object program will contain the instruction

DOS(f[0])

The first formal location must be filled with such an instruction that after return to the body
F contains the value required
A and B are unchanged. (A will contain the local reference point and B will contain the stack pointer).
The return value of S, however, is immaterial.

Left hand value

This has to be evaluated when assignment has to be performed to a formal variable of type arithmetic. On account of possible side effects the proceedings will be as follows:

First a left hand value will be evaluated on top of the stack, then the expression will be evaluated in the F-register, and finally this value will be assigned according to the data on top of the stack. A left hand value is a generalization of the concept "address".

The evaluation of the left hand value is commanded by the object program by

DOS(F[1])

Upon return in the body the return values of S and F are immaterial, A again contains the local reference point, as it did before, and B is increased according to the size of the left hand value, seeing to it that again it points to the first free place.

The left hand value must be such that, when the value to be assigned has been placed in the F-register and the left hand value is indeed the top element, assignment can be effected with

DO(MC[-1])

After this operation the return value of A is still the local reference point, the contents of S are immaterial, B is decreased in accordance with the size of the left hand value processed and F is unaltered if assignment to a real has taken place. If, however, assignment to an actual integer has taken place, then F will contain the rounded value.

Checking of equal types in multiple assignment

In a multiple assignment, all left hand sides must be of the same type. If no formal left hand side is in the list, the checking will be done at translation time. If one or more of the left hand values are formal the check for type consistency will be performed at run time as part of the assignment statement.

As soon as all left hand sides have been evaluated on top of the stack they will be inspected all - including the left hand sides of non formal arrays -; to do this, three system entries are introduced:

SE7: check consistent left hand type
SE8: check integer left hand type
SE9: check real left hand type.

They are of the form of SUBCD instruction; they are preceded by

S:= number of left hand values to be inspected.

SE7 has to be used in a multiple assignment, all elements of which are formal. In the case of one or more non-formal left hand sides the translator can have checked the consistency of the non-formal ones, and use SE8 or SE9, just as the case may be.

The two orders, just generated may not be separated by a program segment transition.
Upon return from SE7, SE8, and SE9 - if the check was satisfactory - A must contain the local reference point, B must have its previous value, the return values of S and F are immaterial. (The assumption here is, that checking will be done after evaluation of the left hand value(s) but before that one of the right hand value.)

Remark 1. The number of left hand values is given explicitly; the alternative - derival from WP - would disable us to carry out this check at the so-called "intermediate assignment".

Remark 2. The possibility to perform this consistency check at procedure entry has been rejected on three grounds:
1) it would impose a scanning burden on the translator, (not serious)
2) it would require the formal locations to distinguish between integer and real (what is otherwise unnecessary)
3) multiple assignment is unimportant anyhow, still more so with formal left hand sides.

Other types of formal parameters will be dealt with later.

Complex arithmetic

This will be described now, now for two reasons: it is not obvious, and we have hardly paid attention to it. Also it has a bearing on the actual formal correspondence, when a actual of type arithmetic may be supplied for a complex formal.

A complex number will occupy in store four consecutive words, two for the real part and two for the imaginary part.

In accordance with our attitude towards complex numbers, we shall implement them as straight forward as possible, only trying to gain, where the gain can easily be won.

The straight forward method puts all results on top of the stack, four words for each operand and we only operate on top of the stack.

The first thing we decide is that, in the case of a simple left hand operand and a complicated expression at the right hand side, we shall do an interchange whenever linguistically possible. (This analysis will be done for arithmetic variables anyhow. So this cannot produce great additional difficulties.)
Up till now the reason to do so is saving of stack space. The next question is under what circumstances we intend to save time as well.

The only operations worth bothering about are the fetch, the assignment, the addition, the negation and the multiplication.

The basic remark is that as long as addition, negation, assignment and fetch are concerned, we can perform our operations first on the real parts and then on the imaginary parts. All primaries being simple

"a:= b + c"

becomes then

F:= real part b; F:= F + real part c; real part A:= F;

F:= im part b; F:= F + im part c; im part a:= F;

This is even OK if a coincides with b or c or both. It becomes more difficult if we have a product, because then there is no non-interference anymore between real and imaginary parts.

If I am not mistaken the following algorithm holds. If a complex product has to be formed, investigate if any direct additions can be performed on it. This defines which anonymous locations will accept the result (if any). Make a list of destinations (including an anonymous one in the case of further processing). Check, whether the ~~fxix~~ first factor occurs among the destinations; if so, save the real part. Check, whether the second factor occurs among the destinations, if so, save its real part. Finally produce the code for the real part computation and store, secondly create the code for the imaginary part, being sure to use the saved real parts. It should be able to cope with

a:= b:= a * b !

If a non-complex primary is used as argument for a multiplication or addition, we can exploit this. This will not be too hard.
This being settled we can resume the System Entries for the complex arithmetic.

SE10 complex division.

This will be a SUBCD instruction. The two top elements of the stack must be complex values, the lower one is replaced by the quotient of the lower one divided by the higher one.

Return value of B is 4 lower then the entry value, A must be unchanged, the return value of S and F is immaterial.

SE11 complex power.

The top element of the stack must contain the complex base, F must contain the exponent value. SE11 starts to check, whether this value is indeed integer and replaces the base by the power. Return value of A and B unchanged, of S and F immaterial.

Complex assignment

This only occurs in the case of separately evaluated complex left hand values, derived from formal or subscripted (complex!) left hand sides. The top of the stack will contain a complex value, underneath will be a left hand value. For every assignment the object program will contain

DOS(MB[-5]).

This will perform the assignment, remove the left hand value, shift the complex value down and decrease B accordingly. Upon return, A must contain the local reference pointer, the values of S and F are immaterial. After the last assignment the x object program contains

B:= B - 4

in order to get rid of the complex value.
(The last transportation is unnecessary, but I think I prefer it that way)

Procedure entry and exit

Type procedures will leave their result on top of the stack.
We shall first treat the non-formal case.
The calling sequence for a procedure of type integer, real, complex respectively starts with "B+1", "B+2", "B+4" respectively.
This has also to be done in the case that the type procedure is called as statement; the way in which the decrease of B (in order to reject the unwanted result) will be effected will be described later.

On top of the place left open the calling sequence will put the formal locations, if any. On top of those, the calling sequence will put the return information.

The stack picture at procedure call will be

```
M[B-10]   :f[0]
          :f[1]
          :f[2]      } formal locations of last actual
          :f[3]      } parameter, if present
M[B-6]    :4*times number of actual parameters +6
          :invariant return address
          :return D
          :return specifier

M[B-2]    :invariant starting address of the procedure
M[B-1]    :context D
 B →      .....        places reserved for the standard locals.
          .....
          .....
```

The present value of B will be local reference point of the fictitious block of the procedure, see below.

We have the following return specifiers

SE12 Normal return from explicit procedure.
This will result in a AP-value equal to A - MA[-6], when A is the local reference point of the fictitious block.

This will be the return specifier in all non-formal calls, except when explicit type procedures are called in the statement situation.

SE13 Reject integer result.
This will result in a AP-value equal to

A - (MA[-6]+1).

This will be the return specifier when an explicit integer procedure has been called in the statement situation.

SE14 Reject real result.
This will produce AP = A - (MA[-6]+2);
it will be used, when an explicit real procedure is used in the statement situation.

SE15 Reject complex result.
This will produce AP = A - (MA[-6]+4);
it will be used when an explicit complex procedure is used in the statement situation.

The formal callsare classified according to the information available at translation time of the body.

If a formal procedure is specified non type it may only be called in statement situation. All types of procedures are admissable as actual one. The calling sequence will start with B:= B + 4 and SE15 will be used as return specifier.

If a procedure is specified as arithmetic, the only actuals acceptable on account of this specification are those of type integer and real.

If it is called in the statement situation, its calling sequence will start with "B+2" (to give room for a result) and the return specifier will be SE14.

If it is called in the expression situation, its calling sequence will start with:

```
F:= 0
MC[0]:= F;
```

the return specifier used will be:

SE16 Make real result.

This will produce the return value of AP equal to A - MA[-6]; but before that, it will check, whether d15 of the lower result word equals d26 of the higher one; if not, -0 will be filled in the lower result word. (To cater for the case that the arithmetic actual was integer and has produced a negative result).

If an procedure is specified as complex, the only actuals acceptable are those of types integer, real and complex.

If it is called in the statement situation, its calling sequence starts with "B+4" and the return specifier will be SE15.

If it is called in the expression situation, the calling sequence will start with:

```
F:= 0
MC[0]:= F;      } filling in an imaginary part = zero

MC[0]:= F;        prepare place for real part
```

and the return specifier used will be SE16.

Remark. Complex numbers will be represented, for this reason by imaginary part followed by the real part. We shall do this consistently. (In contradiction to a previously stated convention.)

A complication arises because a single procedure identifier may be a complete expression, it may be given as actual parameter where the corresponding formal one has been specified arithmetic.

When a formal parameter has been specified arithmetic, the procedure will accept an arithmetic procedure identifier as actual. It will change, however, the formal locations in such a way that DOS(f[0]) will produce a right hand value in F and DOS(f[1]) will produce an alarm (no left hand value).

This checking will change f[0] in such a way that
a) two words on the stack are left open for the result, as the arithmetic procedure might be of type integer, +0 will be filled in (at least in the lower word).
b) the proper calling sequence for a procedure without parameters will be generated. This implies the construction of an invariant return address!
c) as return specifier it will place

SE17: Real value in F.

This one is a combination of SE16 and SE14.
it starts by making a decent real result of the contents of the locations left free. After that it will reject them, but it will return with the value in the F-register.

The fictitious block

As described above the calling sequence prepares in its anonymous space place for the result (if any), the formal locations (if any) and the return information. This information becomes the "nonymous" information of the fictitious block, On top of the return information sufficient room must be available for the standard locals (WP,SV-chain, etc.). The AP-value at the moment of entry will act as the local reference point of the fictitious block.

The text of the procedure body starts with the fictitous block entry consisting of three orders (not separated by program segment transition):

A:= block height of the fictitious block

S:= maximum block height inner block

SE18 Create Display.

SE18 is a SUBCD instruction. Upon return the values of S and F are immaterial. A contains the local reference pointer, B will contain the stack pointer. At this moment the value of the stack pointer will be equal to that of the WP of the fictitious block; this value will also be stored at MA[0], being one of the standard locals.

The task of SE18 is roughly the following.
It has to find place for a new display on top of the stack, the size of this display being desirable from S. If the amount of space in the current stack page is insufficient a new stack page has to be initiated.

The number of places needed consecutevely will be 3 more than the entry value of S, for the Display to be created will have the following formal:

```
          D[-3] : PDC = Parameter Depth Counter
          D[-2] : FBH = Fictitous Block Height
          D[-1] : CBH = Current Block Height
D →       D[0]  : global reference point
          D[1]  : reference point block height 1
              ↓ etc.

          D[S_entry]
B →       .........
```

(The number of places needed consecutively will be a little more than $S_{entry} + 3$, because we must still be able to give some SUBCD-instructions).

SE18, Create Display will set the new PDC equal to 0, FBH and CBH equal to A_{entry}, it will fill the places D[0]...D[CBH-1] with the corresponding elements of the context display, D[CBH] with the local reference point of the fictitious block.
After this preparation the fictitious block has been duly entered: in extending the stack the fictitious block behaves as any other block.

The next three orders check the number of parameter supplied.
Unseparated the text will continue with

```
S:= MA[-6]
U,S - "(4*number of formal parameters +6)",Z
N, SE19
```

where SE19 is an alarm exit for "wrong number of parameters".

After this check the procedure will investigate the actual parameters.

Actual parameterm investigation is done by two unseparated instructions per parameter, viz.

```
S:= f[0]   (with absolute MA-addressing)
```

followed by one of the following system entries (all SUBCD-orders):

SE20	Check formal arithmetic
SE21	Check formal boolean
SE22	Check formal arithmetic array
SE23	Check formal boolean array
SE24	Check formal procedure
SE25	Check formal arithmetic procedure
SE26	Check formal boolean procedure
SE27	Check formal label

SE28 Check formal switch

SE29 Check formal string

SE30 Check formal complex

SE31 Check formal complex array

SE32 Check formal complex procedure

After return - when the check is satisfactory - S and F have immaterial values. A and B must contain the usual information.

If the value list contains scalars, the fictitious text will continue with

S:= 16 (or something else, this is a translator question)

SE1

in order to reserve anonymous space for formal scalar evaluation (case of simple implicit subroutine).

For each scalar in the value list it will contain the code sequence asking for the right hand value followed by an assignment, overwriting the formal locations (this can both be done with MA-addressing).

If arrays by value are to be processed they must be processed now. (see later) This may cause a WP-increase, but there is nothing wrong with that.

The whole picture of a procedure text that will be

```
┌ Create Display (SE18)
│ Check Number of Parameters (SE19)
│ Check Formal Parameters (SE20-SE32)
│ Process Value List (SE1)
│       ┌Block Entry (SE0)  ┐
│       │        :          │   Translation of the body proper
│       │        :          │
│       └Block exit  (SE6)   ┘
└ DO(MA[-3])
```

The block exit SE6 annihilates the block entry SE0; if the procedure body does not contain any local variables, which need space reservation in the stack, this bracket pair may be omitted.

The fictitious block entry Create Display (SE18) is annihilated by the last instruction: an execute of the return specifier supplied at the call.

The Parameter Depth Counter PDC

The following applies to the non-simple implicit subroutines NSIS. When counting the maximum inner block height, the NSIS counts for an increase equal to 1.

When in implicit subroutine is called, this is done under control of a so-called context D. Before entering the NSIS proper the PDC of the context D is increased by one; upon return from the NSIS it is again decreased by 1. As an NSIS might contain a call of another procedure supplying again a NSIS as actual parameter, PDC might get larger than 1.

The Non Simple Implicit Subroutine NSIS

The NSIS is treated as a special kind of procedure with a "parameter block" of a height 1 higher than the calling sequence.

The fact that an actual parameter is a NSIS is given in the formal locations. Its block introduction, however, needs no parameters from the text od NSIS. (The implicit subroutine has no local variables.) Before entering the NSIS, therefore, its context D is already the current one, the PDC and the CBH associated with it have been increased, a local reference point has been introduced and a WP has been set.

The NSIS starts by asking for anonymous space in the usual manner (with SE1).
At the end it will contain the return from the implicit subroutine, which will find its return information under control of the local reference pointer. As part of the return operation PDC and CBH, will be decreased by 1. Further details later.

List of system entries introduced

SE0	Block Entry
SE1	Anonymous Reservation
SE2	Create integer arrays
SE3	Create real arrays
SE4	Create boolean arrays
SE5	Create complex arrays
SE6	Block Exit
SE7	Check consistent left hand type
SE8	Check integer left hand type
SE9	Check real left hand type
SE10	Complex division
SE11	Complex power
SE12	Specify normal return
SE13	Specify return, rejected integer
SE14	Specify return, rejected real
SE15	Specify return, rejected complex
SE16	Specify return, integer to real extension
SE17	Specify return, result in F
SE18	Create Display
SE19	Alarm exit wrong number of parameters
SE20	Check formal arithmetic
SE21	Check formal boolean
SE22	Check formal arithmetic array
SE23	Check formal boolean array
SE24	Check formal procedure
SE25	Check formal arithmetic procedure
SE26	Check formal boolean procedure
SE27	Check formal label
SE28	Check formal switch
SE29	Check formal string
SE30	Check formal complex
SE31	Check formal complex array
SE32	Check formal complex procedure.